

# 電気情報工学科 C 課程第三学年ハードウェア実験資料

2020 年 7 月 1 日:第 7 版



## 改訂履歴

- 2020 年 6 月 2 日: 第 1 版
- 2020 年 6 月 3 日: 第 2 版: 細かな修正
- 2020 年 6 月 8 日: 第 3 版: 2 章の Verilog-HDL テンプレート記述の誤り訂正
- 2020 年 6 月 17 日: 第 4 版: KAPPA3-RV32I の仕様修正  
ジャンプ命令のベースとなる PC アドレスが +4 される仕様だったのを PC に訂正.  
この変更に伴って PC の値を +4 する操作が WR フェイズに移動  
レジスタファイルに書き込む値が PC から PC + 4 に変更
- 2020 年 6 月 22 日: 第 5 版: サンプルプログラム修正  
Kappa3 仕様書のタイポ修正
- 2020 年 6 月 24 日: 第 6 版: 細かな語句の修正
- 2020 年 7 月 1 日: 第 7 版: 付録 C:フェーズ動作の文言修正



# 目次

第 1 章	はじめに	1
1.1	実験の目的および概要	1
1.2	実験のスケジュール	1
1.3	レポート作成上の注意	2
1.4	本手引書の構成	3
第 2 章	Verilog-HDL を用いた論理設計	5
2.1	実験概要	5
2.2	課題 1	6
2.3	課題 2	7
2.4	課題 3	8
2.5	課題 4	9
2.6	課題 5	11
2.7	第 1 回レポート課題	12
第 3 章	マイクロプロセッサの設計	13
3.1	全体の日程	13
3.2	作業手順	13
3.3	サンプルプログラム	17
3.4	第 2 回レポート課題	22
第 4 章	KAPPA3-RV32I 上でのプログラム開発	23
4.1	全体の日程	23
4.2	C 言語プログラムで hello world	23
4.3	入出力	25
4.4	タイマ割込み	30
付録 A	Verilog-HDL 文法リファレンス	33
A.1	概要	33
A.2	簡単な文法規則	34
A.3	モジュール記述	38
A.4	assign 文	39
A.5	演算子	40
A.6	if 文	43

A.7	case 文と casex 文 . . . . .	44
A.8	function 文 . . . . .	45
A.9	always 文 . . . . .	48
A.10	インスタンス記述 . . . . .	53
A.11	Verilog-HDL 記述の注意点 . . . . .	55
付録 B	FPGA ボードの仕様	57
B.1	概要 . . . . .	57
B.2	MU500-RK, MU500-7SEG 用表示ドライバ . . . . .	63
付録 C	教育用プロセッサ KAPPA3-RV32I の仕様 Ver. 0.1	65
C.1	概要 . . . . .	65
C.2	命令フォーマット . . . . .	66
C.3	アドレッシングモード . . . . .	67
C.4	命令セット . . . . .	68
C.5	特権命令と割り込み処理 . . . . .	75
C.6	構成 . . . . .	79
C.7	フェイズ . . . . .	82
C.8	KAPPA3-LIGHT 用の入力・表示モジュール . . . . .	84
C.9	各部の仕様 . . . . .	87
参考文献		105

# 第 1 章

## はじめに

### 1.1 実験の目的および概要

本実験では、最終的なゴールとして単純なアーキテクチャを持つ CPU の設計・製作を行なうことを目的としている。そのために必要な知識および技術を身につけるために下記の事柄について学習を進める。

1. 簡単なデジタル論理回路の設計および実装
2. 計算機アーキテクチャと論理回路・電子回路との関連
3. CAD を用いた論理回路設計
4. FPGA(Field Programmable Gate Array) を用いた動作確認

本実験で扱う CPU は計算機の基本的な構造と機能を学習するのに必要な程度のアーキテクチャと命令セットを持つ 32 ビットの CPU である (KAPPA3-RV32I)。本実験では、FPGA を搭載したボードを用いてこの CPU の動作を把握した上で、これと同じ動作をする論理回路をハードウェア記述言語を用いて各自で設計する。そして、設計した回路を実際に先の FPGA ボード上で実装してその動作を検査・確認する。なお、これらに先立って最初に、論理回路設計について基礎的な技術を習得するために、ごく簡単なデジタル回路の設計・製作を行う。

### 1.2 実験のスケジュール

本実験は概ね下記のスケジュールに従って行う。なお、各テーマの終了ごとにレポートを提出すること。

#### ● ハードウェア実験 1

1. ハードウェア記述言語を用いた論理回路設計 (7 回) 教育用計算機の CAD ツールを用いて以下のよう  
に論理回路設計を行う。
  - － ハードウェア記述言語 (Verilog-HDL) を用いていくつかの論理回路の設計を行う。
  - － ハードウェア記述言語で設計された論理回路を CAD を用いて論理合成し、FPGA ボードに  
ダウンロードすることによって実際に動作させる。

#### ● ハードウェア実験 2

1. CPU の動作理解 (3 回)  
実際に設計する CPU を FPGA ボードにダウンロードし、動作を確認する。特にフェーズごとの  
データの流れと各モジュールが動作するタイミングに注意する。
2. CPU 回路の設計・動作確認 (12 回程度)  
各々のモジュールについて Verilog HDL で設計を行う。各モジュールごとに設計の指針を説明す

るので、これに留意して各人が設計を行う。設計した回路をFPGAへダウンロードし、使用どおりに動作するかチェックを行う。

#### ● PBL-II

##### 1. 制作したCPU上でのプログラムの開発(3回)

開発したCPU上で動作するプログラムをC言語で開発する。また、メモリマップトI/Oや割込みを含むプログラムを記述する。

##### 2. PBL(6回)

制作したCPU上で動作するプログラムにより入力やLEDを活用して、ゲームやデモを作成する。最終日には成果を発表する。

## 1.3 レポート作成上の注意

本実験にのみに関わらず、実験レポートには満たすべき要件がある。それらの点に留意の上、レポートを作成し、提出すること。

#### 1. レポートの提出期限を厳守すること。

レポートを必ず提出すること。レポートが一つでも欠けていれば単位を修得することはできない。この実験に関連する単位は卒論着手要件になっているものがあるので注意すること。また、提出期限を必ず守ること。レポートは必ず提出期限が提示される。期限を守っていないレポートは、たとえ内容が優れていても、それなりにしか採点されない。さらに、提出が大きく遅れた場合は採点できず、そのまま不可となってしまう可能性がある。

#### 2. 何の目的で、どのような実験を行ない、どのような結果を得、どのような設問に答え、それによりどのような考察を得たか、が明確に判ること。

以上の事柄が、第三者に対しても明らかに判るように述べてあること。ここで第三者とは教官ではなく、ハードウェアに関する一般的な知識は持っているが、このカリキュラムについてはなにも情報を持っていない者である。何の説明もなしに、得られた結果のみを出し、どのような問に答えたのかも判らぬ計算結果を書き、何の考察も述べられていないようなレポートは最低の評価しか与えられない。もし、実験を通しての評価が十分でない場合は不可となる可能性もある。

#### 3. 設問に対する回答にも考察をつけること。

設問に対して計算をし、数値を求めたり、本を調べてその内容を書き写すだけならば小学生にもできる。求められているのは、その設問に答えることでどのような考察を得たか、である。

#### 4. 図やグラフ、表にはタイトルおよび説明を付与すること。

タイトルもついていない図、各軸が何を表しているのか書いていないグラフなどは何を意味するのか全く不明である。図、表、グラフにはタイトルおよび必要とされる情報が記入されていなければならない。

#### 5. 他人のレポートとの差別化を図ること。分からないことを他人に聞くことは大切である。しかしながら、その際に重要なのは、教えられたことを理解し、それを自分の言葉で説明できることである。たとえ得られた結果が同じであったり、同一の班に所属しているとしても、他人のレポートの丸写しでは、レポートを提出する意味は全くない。



## 1.4 本手引書の構成

各章の構成は以下のようになっている。

2 章 Verilog-HDL を用いた論理設計．ハードウェア実験 1 の作業についての説明およびレポート課題．

3 章 マイクロプロセッサの設計．ハードウェア実験 2 の作業についての説明およびレポート課題．

付録 A Verilog-HDL 文法リファレンス．

付録 B 実験用 FPGA ボード MU500-RXSET01 の説明書．

付録 C 教育用マイクロプロセッサ KAPPA3-RV32I 仕様書．



## 第 2 章

# Verilog-HDL を用いた論理設計

## 2.1 実験概要

以下の課題 1～課題 5 について，Verilog-HDL を用いて回路の記述を行ない，CAD ツール (Quartus) を用いて論理合成した後，設計データ (sof ファイル) を FPGA ボードにダウンロードして動作を確認せよ。

なお，すでに設計済みの回路の Verilog-HDL ファイルは `public` フォルダにある．設計対象の Verilog-HDL ファイルのスケルトン (中身のないファイル) は `templates` にあるのでそれを利用すること．


### 2.1.1 Quartus の使用方法

本実験では FPGA 用の開発ツール Quartus を用いる．以下では Quartus の使用方法を簡単に述べる．  
`docs/MU500-manual/MU500-RXSET01/1_ユーザズマニュアル/2_FPGA 設計ツール操作手順書` も参照すること．Quartus では「プロジェクト」と呼ばれる単位で設計データを管理する．そこで，以下の手順に従ってプロジェクトを作成する．

1. `File -> New Project Wizard` メニューを選ぶ．
2. 次の画面に以下の情報を入力する．
  - プロジェクトのデータを格納する場所．
  - プロジェクト名
  - Verilog-HDL のトップレベルモジュール名

厳密なルールではないが，プロジェクト毎に個別のフォルダ (ディレクトリ) を用意したほうがトラブルが少ない．このディレクトリと Verilog-HDL のファイルの場所は異なってもよい．ここで指定したフォルダが存在しない場合には自動的に作成される．また，プロジェクト名とトップレベルモジュール名は同一にしておいたほうが間違いが少ない．本実験では全てのトップレベルモジュールを用意しているので，その名前を使用すること．例えば課題 1 なら `kadai1` がトップレベルモジュール名である．

3. 次の画面では使用するファイルを追加が行えるが，後でも追加できるので何も指定せずに `Next` をクリックしてもよい．あらかじめ与えられたファイルがある場合にはここで追加してもよい．
4. 次の画面で使用する FPGA デバイスの指定を行う．ここでは `EP4CE30F23I7` を選択して `Next` をクリックする．中央のリストの絞り込むために，`Device family`, `Package`, `Pin count` などの情報を入力する．
5. 次の画面では使用する外部ツールの登録を行うが，ここではなにもせずに `Next` をクリックする．
6. 確認画面で `Finish` をクリックする．
7. 以上でプロジェクトの作成は終了．

8. **Assignments** -> **Import Assignments...** メニューを選び、ダイアログに `verilog-src/templates/topmodule.qsf` を入力する。これで Verilog-HDL のトップレベルモジュールの入出力ポートと FPGA ボードの各種入出力ピンの接続が行われる。
9. Verilog-HDL ファイルの作成が終わったら **Project** -> **Add/Remove files in project...** メニューを選び、ファイルの追加を行う。
10. 上部ツールバーの  ボタンをクリックしてコンパイルを行う。
11. 下部のメッセージログにエラーが出力されていなければ (エラーは赤字で出力される) コンパイルは成功。青字の warning が有っても FPGA データは正常に作成されているが、不具合の原因が隠されている場合もあるのでデバッグ時には参考にするといよい。
12. FPGA ボードを PC の USB ポートに接続し、電源を入れたあとで、**Tools** -> **Programmer** メニューを選択し、現れたダイアログの **Start** ボタンを押す。その際に左上の **Hardware Setup..** 欄が **USB-Blaster[USB-0]** となっていることを確認すること。正常に書き込みが行われれば右上のプログレスバーが緑色になって中に **100%(Successful)** と表示されるはずである。失敗した場合には接続を確認して再度試すこと。
13. ファイルを修正した場合には 10. 以降を繰り返す。ファイルを追加する場合には 9. に戻る。
14. 既存のプロジェクトで作業を始める場合には最初に **Files** -> **Open project...** メニューをえらび、ファイルの追加や修正を行えばよい。
15. Quartus 中でファイルをエディットする場合には **view** -> **Project navigator** を選び、左上部の **Project Navigator** の種類を **Files** にするとこのプロジェクトで使用しているファイル一覧が現れるのでそれをクリックすると中央にファイルの内容が現れる。
16. Quartus 以外の汎用のテキストエディタを用いてファイルの作成/修正を行ってもよい。ただし、コメント中に日本が含まれている場合にはファイルの文字コードを UTF-8 にすること。

## 2.2 課題 1

4 ビットの信号を 4 桁の 2 進数 (`2'b0000` – `2'b1111`: 0 – 15) とみなして、7 セグメント LED に数字を表示させるための 8 ビット信号 (うち 1 ビットはドット用) を出力する組合わせ回路を Verilog-HDL で記述せよ。7 セグメント LED のパタンは図 B.3 を参考にせよ。

7 セグメント表示用デコード回路のテンプレート

```
module decode_7seg(input [3:0] in,
    output [7:0] out);
    ...
endmodule
```

[ヒント] function 文と case 文を用いること。

`decode_7seg` の設計が終わったら、`kadai1.v`, `led_driver.v` を追加ファイルにしてプロジェクト `kadai1` を作り Quartus でコンパイルせよ。ピン割り当ては `topmodule.qsf` を import すること。

ロータリスイッチの値に応じて 7SEG-LED の表示が正しく行われることを確認せよ。

## 2.3 課題 2

4 ビットの加算器を Verilog-HDL で記述せよ.

4 ビット加算器のテンプレート

```
module add4(input  [3:0] a, b,  // 4 ビット入力 x 2
            input    cin,    // キャリー入力
            output [3:0] s,   // 4 ビット出力
            output    cout); // キャリー出力

...

endmodule
```

[ヒント] そのままの記述が図 A.2 にあるが, できればこれをそのまま使わずに ‘+’ 演算子と連結演算子のみを用いる記述を作ること. 要はキャリー出力を 5 ビット目の和と見なせば良い.

add4 の設計が終わったら `kadai2.v`, `led_driver`, `decode_7seg.v` ファイルに追加したプロジェクト `kadai2` を作り Quartus でコンパイルせよ.

2 つのロータリースイッチと右上隅のプッシュボタンに応じて表示が正しく行われることを確認せよ.

## 2.4 課題 3

4 ビットアップダウンカウンタを設計せよ。

4 ビットアップダウンカウンタのテンプレート

```
module udcount4(input      clock, // クロック信号
                input      reset, // リセット信号
                input      ud,    // 0: アップ, 1: ダウン
                input      enable, // カウントイネーブル信号
                output [3:0] q,    // カウント値の出力
                output      carry); // キャリー出力
...
endmodule
```

[動作] 基本的には `clock` に従ってカウント値を 1 つずつ増減するカウンタ。 `reset` が 0 になった時にカウント値を 0 に初期化する。 `ud` が 0 の時にカウントアップを行い、1 の時にカウントダウンを行う。ただし `enable` が 0 の時はカウント値を変更しない。

また、アップモードで 4'b1111 から 4'b0000 へ遷移するときとダウンモードで 4'b0000 から 4'b1111 へ遷移するときに `carry` を 1 にする。

[ヒント] `always` 文, `if` 文, ノンブロッキング代入文を用いる。ただし, `carry` 信号は組合わせ回路で作ること (`assign` 文を用いる)。

`udcount4` の設計が終わったら `kadai3.v`, `led_driver`, `decode_7seg.v` ファイルを追加したプロジェクト `kadai3` を作り Quartus でコンパイルせよ。

ロータリースイッチと右上隅のプッシュボタンに応じて表示が正しく行われることを確認せよ。特にキャリー出力が正しく生成されているかを確認すること。そのため、クロックの周波数がある程度遅くする必要がある。クロックのロータリースイッチの D や E を用いること。

[チャレンジ課題] 余裕があったら `udcount4` を改造して 10 進 1 桁 (0 – 9) のアップダウンカウンタを設計せよ。

## 2.5 課題 4

キー入力エンコーダおよびキー入力バッファを設計せよ。

キー入力エンコーダのテンプレート

```
module keyenc(input  [15:0] keys,
              output   key_in,
              output [3:0] key_val);
...
endmodule
```

keys は FPGA ボードのプッシュ SW(psw\_a0 - psw\_a3, psw\_b0 - psw\_b3, psw\_c0 - psw\_c3, psw\_d0 - psw\_d3 からシンクロナイザ (後述) を経由して接続した 16 ビットの入力信号である (keyenc の設計にはシンクロナイザを考慮する必要はない)。出力は 4 ビットの信号 key\_val と 1 ビットの信号 key\_in で、いずれかのプッシュ SW が押されたとき、key\_in が 1 となり、押された数字に対応する値を key\_val に出力する。key\_in が 0 の場合 (プッシュ SW が押されていない場合) は key\_val の値はなんでもよい (ドントケア)。

プッシュ SW のレイアウトを図 2.1 に示す。なお、keyenc では右の 1 列 ('Clear', '+', '-', '=') のプッシュ SW の値は用いられない (そもそも keys に含まれていない)。

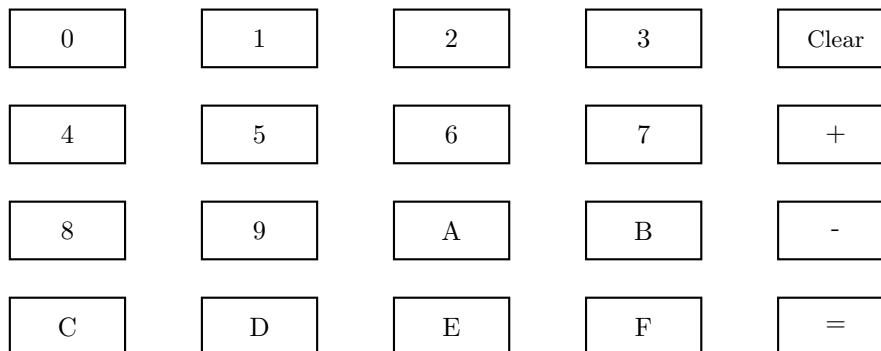


図 2.1 プッシュ SW のレイアウト

[ヒント] keyenc は完全な組合わせ回路である。ただし論理式で記述するのは面倒なので function 文の中で if 文か casex 文を用いる。casex 文の場合には同時に複数のプッシュ SW が押される可能性があることを考えて casex 文のラベルを作成すること。

キー入力バッファのテンプレート

```
module keybuf(input      clock,
              input      reset,
              input      key_in,
              input [3:0] key_val,
              input      clear,
              output [31:0] out);
...
endmodule
```

キー入力バッファは 32 ビットの値を保持するレジスタで、キー入力があったことを知らせる `key_in` とそのときのキーの値を表す `key_val` を入力とする。新しい値が入力されたときには (`key_in == 1`)、以前の値を 4 ビット左に (上位に) シフトして下位 4 ビットに新しい値を入れる。たとえば以前の値が、`32'h0123_4567` で、新たに `key_val = 4'h8` が入力されると以前の値が 4 ビットシフトされて `32'h1234_5670` となり、下位 4 ビットに `key_val` の値が入るので `32'h1234_5678` となる。`clear` が 1 の時は入力バッファが保持している値を `32'b0` にする。

`keyenc`, `keybuf` の設計が終わったら、`kadai4.v`, `led_driver.v`, `decode_7seg.v` と合わせてファイルを追加したプロジェクト `kadai4` を作り Quartus でコンパイルせよ。

[シンクロナイザ]

シンクロナイザのテンプレート

```
module syncro(input  clock,
               input  reset,
               input  in,
               output out);

...

endmodule
```

シンクロナイザはクロックとは非同期に押されるプッシュ SW の信号をクロック信号に同期させるための回路である。さらにシンクロナイザによってプッシュ SW の押下イベントは 1 クロック分のパルスに変換される。シンクロナイザはあらかじめ与えられたものを使うこと。



## 2.6 課題 5

簡単な電卓の設計

今までの課題で作成した回路モジュールを組み合わせて簡単な電卓を設計する。

電卓のテンプレート

```
module calc(input          clock,
            input          reset,
            input [15:0]   keys,
            input          clear,
            input          plus,
            input          minus,
            input          equal,
            output [31:0]  ibuf,
            output [31:0]  cbuf);

...

endmodule
```

内部に 32 ビットの計算結果を保持するレジスタを持つ (計算結果バッファ)。以下のような動作をする。

- リセット時にはすべて 0 にする。
- クリアキー (clear) が押されたら計算結果バッファと入力バッファをクリアする。
- テンキーが押されたらキー入力バッファに数字を入れる (keyenc, keybuf を用いよ)。
- ‘+’ キー (plus) が押されたら現在のキー入力バッファの値を計算結果バッファに転送し、キー入力バッファを 0 にする。
- ‘=’ キー (equal) が押されたら現在のキー入力バッファの値と計算結果バッファの値を足した値を計算結果バッファに代入し、キー入力バッファを 0 にする。

calc を用いたトップモジュールの記述は `kadai5.v` にある。キーと LED の割り当ては以下のようになっている。

- 入力された値は MU500-RK の 8 つの 7SEG-LED に表示する。
- `psw_a4` をクリアキーに割り当てる。
- `psw_b4` を ‘+’ キーに割り当てる。
- `psw_d4` を ‘=’ キーに割り当てる。
- MU500-7SEG の H グループ (8 個) を計算結果の表示に用いる。

[チャレンジ課題] 時間があったら、加算以外に減算 (-) も行えるように改良せよ。その場合、‘=’ が押されたときに直前の演算が + だったのか-だったのかを覚えておく必要がある。

また、16 進ではなく 10 進数として計算、表示するためにはどのようにすればよいか？レポートで考察せよ。

## 2.7 第1回レポート課題

1. 課題1～5について、

- 設計すべき回路の仕様
- Verilog-HDL による設計記述
- 回路記述の簡単な説明

を記せ。チャレンジ課題に関しては余裕があれば答えること。

2. Verilog-HDL を用いて回路を記述する手法を用いない場合には、以下のような手順で手設計を行う必要がある。

- 有限状態機械の状態数最小化
- 有限状態機械の状態符号化
- 状態遷移関数，出力関数の真理値表作成
- 状態遷移関数，出力関数のカルノー図作成
- 状態遷移関数，出力関数の積和系論理式簡単化
- 積和系論理式から論理回路生成

この手順と比べて Verilog-HDL 記述から論理合成を行う手法の得失について考察せよ。特に，上記の手設計の手法が勝っていると思われることはあるだろうか？

3. この実験に対する感想や意見を述べよ。

以上

## 第 3 章

# マイクロプロセッサの設計

### 3.1 全体の日程

表 3.1 に日程を示す。作業内容は目安であり，これより早まっても遅れても構わない。

表 3.1 日程表

	内容
1 回め	ボードの使用法の説明
2 回め	動作理解
3 回め	試問
4 回め	KAPPA3-LIGHT 製作
5 回め	(stconv, ldconv 製作)
6 回め	
7 回め	(phasegen 製作)
8 回め	
9 回め	(debugger モジュールの動作確認)
10 回め	
11 回め	(controller 製作)
12 回め	
13 回め	
14 回め	(動作確認)
15 回め	

### 3.2 作業手順

#### 3.2.1 ボードの使用法の説明および動作理解

- 参考資料を読んで KAPPA3-RV32I(KAPPA3-LIGHT) の内部構造，動作，制御を理解する。
- 見本の設計データを FPGA ボードにダウンロードする。
- サンプルプログラム 1 を機械語に変換する。
- 変換した機械語を FPGA 上のメモリに入力する。
- KAPPA3 を動作させる。

- プログラムの動作を理解する.
- 各命令の各フェイズで内部のレジスタがどのように変化しているかを観測する.
- 各命令の各フェイズで各制御信号がどのように変化しているかを考察し, 表 3.2 のような表を作成する.
- 制御信号のなかには, どの値でも構わない (ドントケア) 場合がある. その場合には ‘-’ と記入せよ.

表 3.2 制御信号を表す表

命令	フェイズ	PC_SEL	PC_LD	MEM_SEL	MEM_READ	MEM_WRITE	MEM_WRBITS	IR_LD	RD_SEL	RD_LD	A_LD	B_LD	A_SEL	B_SEL	ALU_CTL	C_LD
lui x5, 12345h	IF															
	DE															
	EX															
	WB															
addi x5, x5, 678h	IF															
	DE															
⋮																

機械語の生成方法

ここでは, lui x5, 12345h を例にとって機械語の生成方法を説明する. lui 命令は U-type の命令で, [31:12] が即値, [11:7] が  $r_d$ , [6:0] がオプコードで 7'b01110111 となっている.  $r_d$  が x5 なので [11:7] には 5'b00101 が入る. 即値は 12345h なので二進数に変換すると 20'b0001\_0010\_0011\_0100\_0101 (Verilog-HDL の表記) となる. 16 進数の 1 桁が 2 進数の 4 桁に対応することに注意. これを連結すると, 32'b0001\_0010\_0011\_0100\_0101\_0010\_1011\_0111 となる.  $r_d$  フィールドとオプコードの区切りが 4 桁の区切りと合わないことに注意. 最後にこの 2 進数を 16 進数に変換すると, 32'h1234\_52B7 (アセンブリ表記では 123452B7h) となる. KAPPA3-LIGHT では値は 16 進数として入力するので, この 123452B7h を入力する (末尾の h は入力しない).

プログラムの入力方法

1. FPGA ボードの DIP-A[0] を入力モードにする.
2. FPGA ボードの HEX\_A を 8(MAR) にする.
3. プログラムの先頭アドレス (10000000) を入力する. MAR の表示が 10000000 になったことを確認する.
4. FPGA ボードの HEX\_A を 9(memory) にする.
5. 機械語を入力する. 例えば 123452b7 を入力したら Mdr の表示が 123452b7 になっていることを確認する.
6. FPGA ボードの ‘+’ ボタンを押す. MAR の値が 4 加算される.
7. 5 に戻る.

入力を間違えたときは MAR の値を変更するか ‘+’, ‘-’ ボタンでアドレスを変更して正しい値を上書きすれ

ばよい。

### 3.2.2 試問

- サンプルプログラム 2 を機械語に変換する。
- 変換した機械語を FPGA 上のメモリに入力する。
- KAPPA3 を動作させる。
- プログラムの動作を理解する。
- 各命令の各フェイズで内部のレジスタがどのように変化しているかを観測し、表 3.2 のような表を作成する。
- 各命令の各フェイズで各制御信号がどのように変化しているかを考察する。
- 観測結果、考察結果を教官に報告する。

### 3.2.3 KAPPA3-LIGHT の製作

仕様書にしたがって KAPPA3-LIGHT の製作を行う。具体的には以下の手順で設計を行う。

1. ldconv, stconv の制作。

templates/ldconv.v, templates/stconv.v を参考に中身の記述を完成させる。

2. phasegen の制作。

templates/phasegen.v を参考に中身の記述を完成させる。public/phasegen\_test.v をトップモジュールとするプロジェクトをつくり、設計した phasegen.v と public/led\_driver.v public/syncro.v を追加してコンパイルする。ブッシュ SW に応じて正しくフェイズ遷移が行われることを確認せよ。なお、クロック SW は C または D を使用すること。

3. debugger モジュールの確認。

public/kappa3\_light.v をトップモジュールとするプロジェクトを作り、public/debugger.v, public/led\_driver.v, public/kappa3\_light\_core\_dp.v public/mem64kd.v, public/memory.v, public/reg32.v, public/regfile.v, public/syncro.v およびハード実験 1 で設計した keyenc.c, keybuf.v を追加してコンパイルする。public/kappa3\_light\_core\_dp.v は KAPPA3-LIGHT のすべてのレジスタ・メモリを含んだモジュールであるが、制御回路がないのでプロセッサとしては動作しない。ここでは debugger モジュール経由で各レジスタ・メモリに値を書き込んだり、値を表示したり出来ることを確認する。

4. kappa3\_light\_core の作成。

public/kappa3\_light\_core\_dp.v を適当なフォルダ上に kappa3\_light\_core.v という名前でコピーし、このファイルを編集して kappa3\_light\_core の作成を行う。プロジェクトに追加するファイルは以下の通り。

- public/kappa3\_light.v
- public/led\_driver.v
- public/debugger.v
- public/alu.v
- public/mem64kd.v
- public/memory.v

- `public/reg32.v`
- `public/regfile.v`
- `public/syncro.v`
- `keyenc.v` — ハードウェア実験1で作成
- `keybuf.v` — ハードウェア実験1で作成
- `ldconv.v` — 1. で作成
- `stconv.v` — 1. で作成
- `phasegen.v` — 2. で作成
- `controller.v` — `templates/controller.v` をもとにここで作成
- `kappa3_light_core.v` — `public/kappa3_light_core_dp.v` をもとにここで作成

具体的には図 C.2 を参考に `stconv`, `ldconv`, `alu` のインスタンス化を行い、各レジスタ・メモリ間の接続を行う。さらに、制御信号線を出力する制御回路を `phasegen` と `controller` で構成し、インスタンス化する。

5. 全てが完成したら、設計データを FPGA ボードにダウンロードする。表 3.3 に従って各命令の動作チェックを行う。プログラムは全て `10000000h` 番地に入力するものとする。表中の‘機械語’と書かれた列が入力する命令である。入力モードでメモリアドレスを `10000000h` に設定して値を直接入力する。次の‘前条件’と書かれた列に記述されているように各レジスタやメモリの値を入力モードで入力する。その後、実行モードに切り替えて1命令だけ実行を行う (`STEP_INST`)。実行後のレジスタ、メモリの値が‘後条件’の列の内容と等しくなっているかチェックする。なお、明示的には書いていないが、チェック対象のレジスタ・メモリの値は‘後条件’でチェックすべき値と異なる値となるように適当な値をセットしておくこと。例えば、もともとの値が0のレジスタの値が命令実行後に0だったとしても、命令実行の結果0になったのか、それとも変化がなかったのかの区別がつかない。この場合は予め `FFFFFFFFh` などの0以外の値にしておくこと。

## 3.3 サンプルプログラム

### 3.3.1 サンプルプログラム 1

```
10000000  lui  x5, 12345h
10000004  addi x5, x5, 678h
10000008  lui, x6, 98765h
1000000C  addi x6, x6, 432h
10000010  add  x7, x5, x6
10000014  jal  x0, 0h
```

注：プログラム中の 16 進数の値を表すときには末尾に 'h' をつける。たとえば 15h というのは 16 進数の 15 なので 10 進数の 21 である。

### 3.3.2 サンプルプログラム 2

```
10000000 lui  x5, 10000h
10000004 addi x5, x5, 100h
10000008 lw   x6, x5, 000h
1000000C lw   x7, x5, 004h
10000010 blt  x6, x7, 00Ch
10000014 sw   x5, x7, 008h
10000018 jal  x0, 008h
1000001C sw   x5, x6, 008h
10000020 jal  x0, 0h
....
10000100 12345678h
10000104 9ABCDEF0h
```

- このプログラムは何をしているか？
- 10000010 番地の命令を bltu に書き換えたならなにが起こるか？



## 3.3.3 チェックリスト

表 3.3: 動作確認用チェックリスト

ニーモニック	機械語	前条件	後条件
LUI x01, 12345000h	123450B7	pc = 10000000	pc = 10000004 x1 = 12345000
AUIPC x02, 86543000h	86543117	pc = 10000000	pc = 10000004 x2 = 96543004
JAL x03, -100h	F01FF1EF	pc = 10000000	pc = 0FFFFFF04 x3 = 10000004
JALR x04, x05, 100h	10028267	pc = 10000000 x5 = 10000000	pc = 10000100 x4 = 10000004
BEQ x05, x06, 100h	10628063	pc = 10000000 x5 = 0000000A x6 = 0000000A	pc = 10000104
BEQ x07, x08, -100h	F08380E3	pc = 10000000 x7 = 0000000A x8 = 0000000A	pc = 0FFFFFF04
BEQ x09, x10, 100h	10A48063	pc = 10000000 x9 = 0000000A x10 = 00000000	pc = 10000004
BLT x11, x12, 100h	10C5C063	pc = 10000000 x11 = 0000000A x12 = 00000064	pc = 10000104
BLT x13, x14, -100h	F0E6C0E3	pc = 10000000 x13 = FFFFFFFF6 x14 = 00000064	pc = 0FFFFFF04
BLT x15, x16, 100h	1107C063	pc = 10000000 x15 = 00000064 x16 = 0000000A	pc = 10000004
BLTU x17, x18, 100h	1128E063	pc = 10000000 x17 = 0000000A x18 = 00000064	pc = 10000104
BLTU x19, x20, 100h	1149E063	pc = 10000000 x19 = FFFFFFFF6 x20 = 00000064	pc = 10000004

次のページに続く

前のページからの続き

ニーモニック	機械語	前条件	後条件
LB x05, x21, 0h	000A8283	pc = 10000000 x21 = 10000100 M[10000100] = 000000A5	pc = 10000004 x5 = FFFFFFFA5
LB x06, x22, 9h	009B0303	pc = 10000000 x22 = 100000F8 M[10000100] = 0000A500	pc = 10000004 x6 = FFFFFFFA5
LB x07, x23, 6h	006B8383	pc = 10000000 x23 = 100000FC M[10000100] = 00A50000	pc = 10000004 x7 = FFFFFFFA5
LB x08, x24, -1h	FFFC0403	pc = 10000000 x24 = 10000104 M[10000100] = A5000000	pc = 10000004 x8 = FFFFFFFA5
LW x09, x25, -4h	FFCA483	pc = 10000000 x25 = 10000104 M[10000100] = 12345678	pc = 10000004 x9 = 12345678
SB x26, x27, 0h	01BD0023	pc = 10000000 x26 = 10000100 x27 = 0000005A M[10000100] = 00000000	pc = 10000004 M[10000100] = 0000005A
SB x28, x29, 9h	01DE04A3	pc = 10000000 x28 = 100000F8 x29 = 0000005A M[10000100] = 00000000	pc = 10000004 M[10000100] = 00005A00
SB x30, x31, 6h	01FF0323	pc = 10000000 x30 = 100000FC x31 = 0000005A M[10000100] = 00000000	pc = 10000004 M[10000100] = 005A0000
SB x01, x02, -1h	FE208FA3	pc = 10000000 x1 = 10000104 x2 = 0000005A M[10000100] = 00000000	pc = 10000004 M[10000100] = 5A000000
SW x03, x04, -4h	FE41AE23	pc = 10000000 x3 = 10000104 x4 = 5AA500FF	pc = 10000004 M[10000100] = 5AA500FF
ADDI x10, x05, -3h	FFD28513	pc = 10000000 x5 = 0000000A	pc = 10000004 x10 = 00000007

次のページに続く

前のページからの続き

ニーモニック	機械語	前条件	後条件
ADD x11, x06, x07	007305B3	pc = 10000000 x6 = 0000000A x7 = FFFFFFFD	pc = 10000004 x11 = 00000007
SLT x12, x08, x09	00942633	pc = 10000000 x8 = 0000000A x9 = 00000064	pc = 10000004 x12 = 00000001
SLT x13, x10, x11	00B526B3	pc = 10000000 x10 = FFFFFFF6 x11 = 00000064	pc = 10000004 x13 = 00000001
SLT x14, x12, x13	00D62733	pc = 10000000 x12 = 00000064 x13 = 0000000A	pc = 10000004 x14 = 00000000
SLT x15, x14, x15	00F727B3	pc = 10000000 x14 = 00000064 x15 = FFFFFFF6	pc = 10000004 x15 = 00000000
SLTU x12, x08, x09	00943633	pc = 10000000 x8 = 0000000A x9 = 00000064	pc = 10000004 x12 = 00000001
SLTU x13, x10, x11	00B536B3	pc = 10000000 x10 = FFFFFFF6 x11 = 00000064	pc = 10000004 x13 = 00000000
SLTU x14, x12, x13	00D63733	pc = 10000000 x12 = 00000064 x13 = 0000000A	pc = 10000004 x14 = 00000000
SLTU x15, x14, x15	00F737B3	pc = 10000000 x14 = 00000064 x15 = FFFFFFF6	pc = 10000004 x15 = 00000001

以上

### 3.4 第2回レポート課題

1. このプロセッサをパイプライン化することを考える。すると、このままでは命令フェッチとデータメモリのアクセスを同時に実行することはできないため、命令メモリとデータメモリを分離する必要がある(構造ハザードの回避)。それ以外にどのような場合を考慮しなければならないか、また、その問題を解決するためにはどうすればよいか具体的に述べよ。
2. この実験に対する感想や意見を述べよ。

## 第 4 章

# KAPPA3-RV32I 上でのプログラム開発

### 4.1 全体の日程

表 4.1 に日程を示す。作業内容は目安であり、これより早まっても遅れても構わない。

表 4.1 日程表

	内容
1 回め	C 言語のプログラムの入力、コンパイル、シミュレーション、実機実行
2 回め	メモリマップト I/O、ビット操作
3 回め	タイマ割込み

### 4.2 C 言語プログラムで hello world

#### 4.2.1 VirtualBox

RISCV のための C 言語コンパイラ、シミュレータなどの開発環境は、VirtualBox と呼ばれる仮想マシン環境に準備してある。仮想マシン上には Linux(Ubuntu) がインストールされている。仮想マシン環境のことをゲスト、仮想マシンを動作させている PC をホストと呼ぶ。

使用手順は以下の通りである：

1. Oracle VM VirtualBox を起動する。
2. ubuntu\_xenial を起動する。
3. ログイン画面でユーザ名「riscv」を選択し、パスワード「riscv」を入力する
4. ログインできたら、左側のランチャから terminal を起動する

■注意 VirtualBox 上内に作成したファイルは、再起動すると消えてしまう。/mnt からホスト (使っている PC) の D ドライブが見えるので、ここで作業を行う。

#### 4.2.2 C 言語のプログラムを入力する

まず、D ドライブに GitHub からサンプルプログラムやヘッダファイルなどをダウンロードする。GitHub Destkop で <https://github.com/kappa3-rv32i/rv32i-c> clone する。

先ほど clone した D:

rv32i-c に、hello.c を作成し、以下を入力する:

hello.c: 左上の 7 セグ LED を点灯

```
#include "init.h"
#include "encoding.h"

#define MU500      ((volatile unsigned char *)0x04000000)

int main()
{
    // 7seg test
    MU500[0] = 0xff;
    while (1);
}
```

このプログラムは最も左上の 7 セグ LED のセグメントをすべて点灯するプログラムである。このプログラムの詳細は後ほど理解することにした。

### 4.2.3 コンパイルする

今回は開発するコンピュータ（ホスト）と、実行するコンピュータ（ターゲット）の種類が違う。そのため、記述したプログラムからターゲットで動作する実行ファイルを生成するためには、それ専用のコンパイラを使う必要がある。本実験で使用するコンパイラは gcc で、32bit RISC-V プロセッサのためのコンパイラは「riscv32-unknown-elf-gcc」という名前である。

コンパイルは VirtualBox のターミナル上で実行する。まずはホストコンピュータの D ドライブがマウントされている /mnt に現在のディレクトリを移動する。

```
cd /mnt/rv32i-c
```

次にコンパイルしたいが、コンパイラを手動で実行しようとすると煩雑であるため、make [作りたい実行ファイル名] でコンパイルできるように設定してある。作りたい実行ファイル名というのは、ソースコードが記述されている .c ファイルの .c を取り除いた名前である。今回の例では hello.c がソースコードファイル名なので hello になる。

```
make hello
```

### 4.2.4 シミュレータ上での実行する

得られた実行ファイルを実行したいが、実際にターゲットで実行する前に、開発 PC 上で動作確認をしたい。というのも、ターゲット上で実行するために、ターゲットにプログラムをアップロード等する必要があるため、比較的時間がかかるためである。また、入出力機器が限られているため、「printf」すら実行出来ない。今回はターゲットと PC ではプロセッサの種類が異なるためそのままでは実行出来ない。そこで、実行ファイルを PC 上で実行するために命令セットシミュレータを使用する。同時に、MU500 ボードに搭載されている 7 セグ LED やボタンなどをシミュレーションする GUI を使用する。これはプログラムの論理的な振る舞いを確認するだけであれば、使用する必要は無い。シミュレータの起動手順は以下の通りである。これらのシミュ

レータは VirtualBox 上で動作させる:

1. GUI(7 セグ、ボタンシミュレータ) を起動する

```
python mu500_7seg.py
```

2. 命令セットシミュレータを起動する

```
spike -l -d -p1 --isa RV32IMAFDC -m0x10000000:0x00100000 hello
```

3. 命令セットシミュレータ上でプログラムを実行するために「r」と打ち込む
4. 7 セグ LED が点灯することを確認する

#### 4.2.5 実機での実行

T.B.D.

### 4.3 入出力

#### 4.3.1 概要

PC 上のプログラムを開発している時にはあまり意識する必要は無いが、今回のようなハードウェアを直接取り扱うようなプログラムでは、入出力 (Input/Output; I/O) の仕組みをある程度理解しておく必要がある。I/O とは、プロセッサ上で動作するソフトウェアが外部からの入力を受け取ったり、外部へ信号を出力したりすることである。例えば、ソフトウェアがボタンが押されたことを検知したり、LED を光らせたりすることがそれにあたる。

I/O の実現方法にはいくつかあるが、本演習ではメモリマップト I/O(memory-mapped I/O) を使用する。メモリマップト I/O とは、通常のメモリへのアクセスと同様の扱いで I/O 機器を操作することである。プロセッサ、ひいてはソフトウェアがどのようにメモリにアクセスするのかについては、コンピュータアーキテクチャ I の講義資料を参照されたい。

I/O 機器にはレジスタと呼ばれる小規模のメモリが搭載されている。このレジスタはプロセッサ内のレジスタとは別のものであることに注意されたい。メモリマップト I/O の場合にはそのレジスタ群が通常のメモリと同様に扱える。すなわち、レジスタへのアクセスをするための特別な変数があったとすると、その変数への代入がそのレジスタへの代入に、その変数の参照はそのレジスタへの参照となる。(正確には通常の変数の配置アドレスは設定できない(しにくい)ので、ポインタを使用してそのレジスタにアクセスするための変数を定義する)

#### 4.3.2 メモリマップ

下表に本実験環境におけるメモリマップを示す。メモリマップとは、メモリ空間においてあるアドレスの範囲にどの機器が接続されているのかを示す対応表である。この表に従って I/O に関するソフトウェアを記述する。各機器の詳細な説明は次節以降で説明する。

開始	終了	内容
0x0200 0000	0003	CLINT (タイマなど)
0x0400 0000	004b	MU500 (LED など)
0x1000 0000	ffff	RAM (64kbyte)

### 4.3.3 7セグLEDとドットマトリクスLED

■概要 下表に7セグLEDとドットマトリクスLEDのオフセットを示す。オフセットとはメモリマップで示したアドレスの範囲内のどこにレジスタがあるかという情報である。MU500は0x0400 0000に配置されているため、0x0400 0000 + オフセットが実際にアクセスすべきアドレスとなる。例えば最左上の7セグLEDを制御するためのレジスタは、アドレス0x0400 0000 + オフセット0なので、0x0400 0000に配置されていることになる。

開始	終了	内容	
0x00	0x3f	7セグ	8行×8列×8bit
0x40	0x47	ドットマトリクス	8行×8列×1bit

■7セグLED 7セグLEDひとつごとに1バイトのレジスタが割り当てられている。その順番は最左上の7セグLEDがオフセット0、そのすぐ右がオフセット1…と配置されている。また、7セグLEDの各セグメントは下図のように名前がついている。1バイトの内訳はMSB側から1ビットずつ「DP G F E D C B A」となっている。また、各ビットは1のとき点灯し、0のとき消灯する。

例えば1の形に点灯したい場合には、B、Cを点灯させればよいので、2進数で0000 0110、16進数で0x06をレジスタに書き込めば良い。最初に実行したhello.cの例では

```
MU500[0] = 0xff;
```

としている。MU500という変数はレジスタ群をまとめた配列である（正確な説明は後ほど説明する）。このプログラムでは、MU500のオフセット0に0xffを代入するので、最左上の7セグLEDをすべてのセグメントを点灯することになる。

■ドットマトリクスLED ドットマトリクスLEDは8つまとめて1バイトである。最左上の8つがオフセット0x40、その左が0x41、2行目の左側が0x42、…となる。ドットマトリクスLEDのメモリ割り当てを表4.2に示す。実際の配置ではA0が左端で隣がA1、A2の順に並んでいるのでMSBを左側を書く場合には逆順になることに注意。例えば、最左上を点灯、消灯、点灯、消灯…のように交互に光らせたい場合には、2進数で0101 0101、16進数で0x55を0x40に書き込めば良いので、

```
MU500[0x40] = 0x55
```

と記述することになる。



表 4.2 ドットマトリクス LED のメモリ割り当て

オフセット	MSB(7)	6	5	4	3	2	1	LSB(0)
0x40	A7	A6	A5	A4	A3	A2	A1	A0
0x41	A15	A14	A13	A12	A11	A10	A9	A8
0x42	B7	B6	B5	B4	B3	B2	B1	B0
0x43	B15	B14	B13	B12	B11	B10	B9	B8
0x44	C7	C6	C5	C4	C3	C2	C1	C0
0x45	C15	C14	C13	C12	C11	C10	C9	C8
0x46	D7	D6	D5	D4	D3	D2	D1	D0
0x47	D15	D14	D13	D12	D11	D10	D9	D8

#### 4.3.4 入力の取得

入力に関するオフセットを下表に示す。ボタンや DIP スイッチはひとつ 1 ビットわりあてられて 8 つで 1 バイトである。ロータリスイッチはひとつで 1 バイト割り当てられている。

開始	終了	内容	
0x48	0x4b	ボタン	4 行× 5 列
0x4c	0x4d	ロータリスイッチ	8bit x 2
0x4e	0x4f	DIP	8bit x 2

1 行に 5 つのボタンがあり、全部で 4 行の計 20 個のボタンがあるが、きりのよい所で 1 行分を 1 バイトで表す。表 tab:psw にプッシュ SW のメモリ割り当てを示す。各バイトの上位 3 ビットは未使用である。例え

表 4.3 プッシュ SW のメモリ割り当て

オフセット	MSB(7)	6	5	4	3	2	1	LSB(0)
0x48	—	—	—	A4	A3	A2	A1	A0
0x49	—	—	—	B4	B3	B2	B1	B0
0x4a	—	—	—	C4	C3	C2	C1	C0
0x4b	—	—	—	D4	D3	D2	D1	D0

ば左上のプッシュ SW(A0) の状態を知るためにはオフセット 0x48 の最下位ビットの値を調べればよい。ビットごとの操作の方法については後ほど説明する。左上のボタンが押されたときに何かを実行するサンプルコードを以下に示す:

```
if (MU500[0x48] & 0x01) {
// なにかの処理
}
```

MU500[0x48] でオフセット 0x48 の値を取得し、LSB が 1 かどうかを判定するために、ビット演算子「&」で判定している。

以下にボタンが押されたらそれに反応して 7 セグ LED を点灯させるサンプルを示す。

—— ボタンに対応して 7 セグ LED を点灯させるサンプル ——

```
#include "init.h"
#include "encoding.h"

#define MU500      ((volatile unsigned char *)0x04000000)

int main()
{
    while (1) {
        MU500[1] = MU500[0x48];
    }
    return 0;
}
```

#### 4.3.5 ビット操作のテクニック

C 言語のプログラムでは、変数は一番細かい単位でも 1 バイトごとでしか操作できない。しかしながら、I/O を取り扱っていると、より細かい 1 ビット単位での操作が必要となることが多い。ここでは C 言語で 1 ビット単位で値を操作する方法について説明する。

■あるビットの 0/1 判定 あるビットが 0/1 かを判定したい。例えばボタンが押されているかを判断したいなどである。この場合には、判定したいビットだけを 1 にし、それ以外を 0 にした定数を作り、それと対象変数との AND を計算することが多い。

例えば LSB が 1 かどうかを判断するためには、

1. LSB のみを 1 とした定数を作る。2 進数で 0000 0001 なので、これを 16 進数に変換して 0x01 という定数を作る。
2. 対象変数と 1) の定数を&演算する。例えば MU500[0x48] & 0x1 と記述する。
3. ボタンが押されていれば演算結果は 0 以外（この場合は 1）、押されていなければ 0 となる。

なお、C 言語では 2 進数を定数リテラルとして記述できないため、定数リテラルとして記述できる 16 進数に変換してからプログラムを記述する必要がある。2 進数 4 桁で 16 進数 1 桁であるため、この表は頭に入れておくと便利である。

■あるビットを 0 にクリア あるビットを 0 にしたい。ビットを 0 にすることをクリアと呼ぶ。この場合はクリアしたいビットのみを 0 としそれ以外のビットを 1 とした定数と、対象の変数の AND を計算すれば良い。ただし、通常はクリアしたいビットのみを 1 とし、それ以外を 0 とした定数を記述する方が簡単のため、そのような定数の NOT を計算して使用することが多い。例えば LSB を 0 に設定したい場合には、

1. LSB のみを 1 とした定数を作る。2 進数で 0000 0001 なので、これを 16 進数に変換して 0x01 という定数を作る。
2. 1 の NOT を計算する。0x01 で計算できる。
3. 2 と対象変数の&を計算し、元の変数に書き戻す。例えば対象の変数を a とすると、a &= 0x01 とする。

■あるビットを 1 にセット 1 にセットしたいビットのみを 1 にしそれ以外を 0 にした定数を記述し、それと対象変数との OR を計算する。

#### 4.3.6 レジスタへのアクセスの細かい話

メモリマップト I/O のために、機器に搭載されているレジスタへのアクセスをするための特別な変数を準備する、と、冒頭で記述した。実際には変数を指定したアドレスに配置することは少し面倒なので、アドレスと間接参照の演算子を使用して記述する。例えば MU500 にアクセスするための記述は以下の通りである：

```
#define MU500      ((volatile unsigned char *)0x04000000)
```

0x0400 0000 に配置されているため、定数をそのように記述する。C 言語では、型を指定しない定数は整数型として取り扱われるため、キャストを用いて希望の型に変換する。今回の 0x04000000 は、アドレス（ポインタ型）であり、かつ、そのアドレスにある変数（レジスタ）は符号無し 8 ビット整数なので、

```
unsigned char *
```

型に変換する。

また、見慣れないキーワード「volatile」が付記されている。これは、対象変数がこのプログラム以外から書き換えられる可能性があるため、そのつもりでコンパイルしてほしい旨をコンパイラに伝えるための修飾子である。このプログラム以外というのは、ハードウェアや並行に実行されているプログラムがという意味で、別のタスクや後述する割込みハンドラなどのことである。これらの並行実行されるプログラムが変数を変更する可能性がある場合には、volatile の指定が必要となる。また、「そのつもりでコンパイル」というのは、通常 C 言語のコンパイラはコンパイル対象のプログラムが並行性を持たずただひとつのタスクで実行されるという仮定をしてコンパイルを実行する。このときなるべくプログラムが高速に実行出来るように、プログラムを多少書き換えてしまうことがある。このことを最適化と呼ぶ。volatile はこの最適化を抑制し、並行性があるプログラムでの誤動作を防ぐために必要となる。

#### 4.3.7 演習 1

7 セグ LED を 2 桁使用し、ボタンが何回押されたかを 10 進数で表示するカウンタを作成すること。なお、使用する 7 セグ LED やボタンの位置は自由にして良い。

1. プログラム中で指定した 10 進数 1 桁の数値を 7 セグ LED に表示するプログラムを作成する。switch case 文で数値毎に場合分けし、7 セグ LED のレジスタに代入してもよいのだが、10 進 1 桁の数値と、そのときに 7 セグ LED レジスタに代入すべき値の表を配列で保持しておくことで簡単に記述できる。例えばその配列名が dec27seg だったとすると、dec27seg[1] と配列を参照したとき、0x06 が取得できるような配列をあらかじめ作成しておくが良い。
2. 10 進 1 桁のカウンタを作成する。ボタンが押される毎に +1 し、1) を使用して 7 セグに表示する。10 になったら 0 から始める。
3. 10 進 2 桁のカウンタを作成する。

## 4.4 タイマ割込み

### 4.4.1 割込み

割込みとは、プロセッサが周辺機器などから非同期に通知を受け取る仕組みのひとつである。今回の演習では割込みを使用する周辺機器はタイマのみである。

タイマの例で割込みが発生してからの一連の流れを見ていこう。タイマハードウェアはあらかじめ指定した時間経過すると割込みを発生させる。すると、プロセッサは割込みを受け取り、現在実行しているプログラムを中断する。そして、割込みハンドラと呼ばれる、割込みが発生したときに実行される特別なプログラムを実行する。割込みハンドラが特別な命令を使ってその終了をプロセッサに通知すると、プロセッサは元々実行していたプログラムの処理を再開する。

この一連の流れを見て考察すると、割込みに関連したプログラムを記述するときに、考えなければならないことが見えてくる。割込みハンドラは可能な限り短時間で終了しなければならない。というのも、プロセッサは main プログラムとなる元々のプログラムの実行を中断した上で割込みハンドラを実行するため、割込みハンドラの実行時間が長くなると元々のプログラムの実行時間に大きな影響を与えてしまうからである。

割込みハンドラは main プログラムと並行に実行されることを意識してプログラムを記述する必要がある。例えば main プログラムと割込みハンドラで共有する変数は volatile をつけて定義する必要がある。main プログラムから見ると割込みというのはいつ発生するかわからない事象であり、ひいては、割込みハンドラはいつでも実行されうるプログラムであるので、両者は並行に実行されているということに等しい。

割込みされたくない区間は割込みを禁止する。プログラムの途中で一貫して変数を変更したいなど割り込まれると困る区間がでてくる。そのような区間のはじめに割込みを禁止し、必要な処理を実行した上で、割込みを許可する。この区間のことを割込み禁止区間と呼ぶ。割込み禁止区間はできる限り短くしなければならない。適切なタイミングで割込みを処理できなくなるためである。

### 4.4.2 タイマレジスタ

タイマを制御するためのタイマ関連するレジスタを下表に示す。これらのレジスタもメモリマップトである。

レジスタ名	用途
MTIME	Machine Time Register でプロセッサ起動時からの時間が格納されている。
MTIMECMP	Machine Timer Compare Register で、このレジスタの値が MTIME と等しくなったときに割込みが発生する。

あらかじめ MTIMECMP の値を、MTIME より少し大きい未来の時刻に設定しておけば、その時刻に割込みが発生する。

### 4.4.3 サンプルプログラム

以下にタイマ割込みを発生させる単純なプログラムを示す。このプログラムは左上の 7 セグ LED をタイマで点滅させる。

—— タイマ割り込みを発生させるサンプル ——

```
#include "init.h"
#include "encoding.h"

#define MU500_BASE      0x04000000

/* 割り込みが発生した回数を保持するカウンタ */
static volatile unsigned interrupt_count;
/* 何クロックごとに割り込みを発生させるかを設定する定数 */
static unsigned delta = 10;

/* 割り込みハンドラ関数。今回は使用しないので引数の詳細は割愛 */
void *handler(unsigned hartid, unsigned mcause, void *mepc, void *sp)
{
    /* カウンタをインクリメント */
    interrupt_count++;
    /* MTIMECMP は配列で今回は 0 番目を使用する。
       MTIMECMP に将来の MTIME の値を代入する（そのときに割り込みが発生する）
       この操作は毎回必要。よく考えたら当たり前だが。*/
    MTIMECMP[0] = MTIME + delta; /*
    return mepc;
}

int main()
{
    /* カウンタ変数をリセット */
    interrupt_count = 0;
    /* 割り込みハンドラとして定義した handler 関数を設定 */
    set_trap_handler(handler);
    /* 初回の割り込み発生時刻を設定 */
    MTIMECMP[0] = MTIME + 10;
    /* タイマ割り込みを許可 */
    enable_timer_interrupts();

    while(1) {
        if (interrupt_count % 2 == 0) {
            MU500[0] = 0xff;
        }else{
            MU500[0] = 0x00;
        }
    }
}
```

#### 4.4.4 演習 2

タイマを使用して 1 秒ごとにカウントアップし、現在のカウントを 10 進数で 7 セグ LED に表示するプログラムを作成する。また、プッシュボタンが押されたらカウントを 0 に設定する。なお、使用する 7 セグ LED やボタンの位置は自由にして良い。

1. 割込みハンドラを記述する。volatile 指定したグローバル変数を定義する。グローバル変数をカウントアップする。次回の割込み発生時刻を設定する。シミュレーションで変数が変わることを確認する。
2. main 関数を記述する。カウントが入っているグローバル変数を参照し、7 セグ LED を点灯させる。演習 1 の結果を用いてプログラムを記述する。
3. main 関数にボタンが押されたらカウントをリセットするプログラムを追記する。

## 付録 A

# Verilog-HDL 文法リファレンス

本章では Verilog-HDL の文法について簡単に説明する．詳細については参考文献などで確認すること．なお，Verilog-HDL の文法にはオリジナル (Verilog1995) と Verilog2001 と呼ばれる新しい文法の 2 種類が存在する．Verilog2001 はオリジナルの Verilog-HDL の上位互換となっている．ここでは主にオリジナルの Verilog-HDL の文法について述べる．一部 Verilog2001 の文法についても説明する．現在利用可能なツール (プログラム) はどちらの文法も受け付けることが可能である．

## A.1 概要

Verilog-HDL は見かけは C 言語によく似たプログラミング言語のように見えるが，ハードウェアを記述することに特化しているためいくつかの特徴を持つ．

まず，Verilog-HDL の記述はハードウェアを「合成」するための記述と「シミュレーション」するための記述の 2 種類に分類される．合成用の記述はシミュレーション可能であるが，シミュレーション用の記述は必ずしも合成可能ではない．つまり，合成用の記述は「このように作る」という目的にで書いているのに対してシミュレーション用の記述は「このように動作してほしい」という期待する動作を書いているものとなっている．本実験では合成用の記述をメインに説明する．シミュレーション用の記述は必要に応じて説明を行う．

次に，Verilog-HDL では場所によって使用できる構文要素が異なる．これが一般的なプログラミング言語と比べてわかりにくい点である．Verilog-HDL の文法要素は大きく分けて以下の通りである．

- 宣言 (Declaration)
- 要素 (Item)
- 文 (Statement)
- 式 (Expression)

このうち，モジュール (詳細は後述．回路全体を表す) 直下に記述出来るのは宣言と要素のみである．通常のプログラミング要素では当たり前記述できる「文」は実は記述可能ではない．これは Verilog-HDL がハードウェアを記述するために開発されたことに起因するが，慣れないうちは間違いやすいので注意すること．

### A.1.1 構成要素 (Item)

前述の様にモジュール直下には宣言と要素 (Item) のみ記述できる．Verilog-HDL の Item には様々ものがあるがここではよく用いられるものについて説明する．

- assign 文

- function 宣言
- always 文
- init 文
- インスタンス記述

assign 文や always 文などを書いてあるが、厳密な Verilog-HDL の文法ではこれらは Item であって Statement ではない。assign 文はハードウェア中の結線を表す。function 宣言はある意味、組み合わせ論理回路の入出力の仕様を関数の形で記述したものである。ただし、関数宣言を行っただけでは対象の回路は生成されない。回数は式 (Expression:後述) 中で呼び出される必要がある。always 文は定期的に繰り返されるイベントの記述を行うためのものである。もともとはシミュレーション目的で用いられていたが、書き方を制限することで順序回路の合成用記述に用いられる。init 文は一度だけ実行されるイベントの記述に用いられる。これは純粋にシミュレーション用であり合成目的には用いられない。

### A.1.2 文 (Statement)

通常のプログラミング言語の場合はメインの構成要素であるが Verilog-HDL では function, always, init の内側でのみ記述可能である。主な Statement を以下に示す。

- if 文
- case 文, casex 文
- begin — end ブロック

シミュレーション用にはこれ以外にも様々はものが用意されているが、本実験ではこれだけ使えればよい。それぞれの文の詳細は後述する。

## A.2 簡単な文法規則

### A.2.1 識別子

識別子とは信号線や端子、モジュールなどに付けられる名前のことで、ソフトウェアのプログラミング言語における変数名や関数名などと同類の概念である。Verilog-HDL の識別子はアルファベット (a ... z, A ... Z) および数字 (0 ... 9), アンダースコア '\_', ダラ '\$' から構成された文字列で、先頭は必ずアルファベットでなければならない。また、一部の文字列は予約語と呼ばれる特別な意味を持つ語になっているので識別子としては利用できない (例: if や module など)。言語規約上識別子の長さは 1024 文字以内と決められているが、CAD ツールによってはそれよりも短い長さしか識別しない場合もありうる。また、大文字と小文字は区別されるので signal と Signal は異なった識別子を表しているが、これも CAD ツールによっては区別されない場合があるので注意が必要である。一般的に、文法的には正しくても signal と Signal を混同させるような記述は好ましくない。



## 識別子の例

- 正しい識別子

```
abc clk reset$ long_identifier x1
```

- 不正な識別子

```
99input
```

数字が先頭にある.

```
pin@mod1
```

使用不可の文字 (@) を含む.

```
or
```

予約語

これ以外にも`'(バックスラッシュ) から始まる任意の文字列を識別子とみなす規則も存在するが, これは Verilog-HDL 以外の形式のデータを強制的に Verilog-HDL のデータに変換するなどの緊急避難的な目的で導入されたものであり, 可能な限り用いるべきではない.

### A.2.2 予約語

表 A.1 に予約語の一覧を示す。この表に現れる文字列は予約語として特別な意味を持っているので識別子として用いることはできない。

always	and	assign	begin	buf
bufif0	bufif1	case	casex	casez
cmos	deassign	default	disable	edge
else	end	endcase	endfunction	endmodule
endprimitive	endspecify	endtable	endtask	event
for	force	forever	fork	function
highz0	highz1	if	innone	initial
inout	input	integer	join	large
macromodule	medium	module	nand	negedge
nmos	nor	not	notif0	notif1
or	output	parameter	pmos	posedge
primitive	pull0	pull1	pullup	pulldown
rcmos	real	realtime	reg	release
repeat	rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	small	specify	specparam
strong0	strong1	supply0	supply1	table
task	time	tran	tranif0	tranif1
tri	tri0	tri1	triand	trior
triereg	vectored	wait	wand	weak0
weak1	while	wire	wor	xor
xnor				

表 A.1 予約語の一覧

### A.2.3 論理値

信号の値として、通常の “1” および “0” の他に “Z(または z)” と “X(または x)” の 4 つの値を取る。Z はハイ・インピーダンス (その信号に接続したすべての MOS トランジスタがオフになっており、高抵抗値のため電位が定まらない状態) を表す。X は物理的には存在しない値だが、電源投入直後のフリップフロップの値のように 0 か 1 かどちらの値をとるか分からない状態をシミュレートするのに用いたり、casex 文 (後述) で 0, 1 どちらのパタンにもマッチする値として用いる。

### A.2.4 数値

ソフトウェアのプログラミング言語と異なり、Verilog-HDL における数値は値だけでなく、ビット長の情報も持っている。また、数値の表し方に 2 進、8 進、10 進、16 進の 4 通りがある。具体的には Verilog-HDL における数値は、

<ビット長>’ <基数><符合無し数値>

の形式で表される。基数を表す記号は以下の通り。

B または b	2 進数
O または o	8 進数
D または d	10 進数
H または h	16 進数

16 進数の場合、10～15 の数値は A～F(または a～f) で表す。ビット長が省略された場合には 32 ビットとなる。また、ビット長および基数が省略された場合には 32 ビットの 10 進数となる。桁の多い数値を見易くするために ‘\_’(アンダースコア) を適当な位置に挿入することもできる。ただし、コンパイラはアンダースコアを無視するだけなので、アンダースコアを 4 ビットごとなどの適切な個所に挿入するのは設計者の責任である。4 ビットめと 8 ビットめにアンダースコアを入れたつもりが 3 ビットめと 7 ビットだったとしてもコンパイラはなにも言わない。

#### 数値の例

1'b0	1 ビットの 0
4'b1001	4 ビットの 2 進数で 1001(=10 進数の 9)
5'b01XX	5 ビットの 2 進数で下位 2 ビットは不定値
8'h5f	8 ビットの 16 進数で 5F(=10 進数の 95)
156	32 ビット 10 進数で 156
16'b0001_0100_1010_1111	アンダースコアの使用例

Verilog-HDL では負の数という概念がないので、負数を扱うためには補数表現を用いたり、符号用ビットを用意するなど設計側で明示的に工夫を行う必要がある。

### A.2.5 コメント

コメントとは記述されていても無視される文字列のことで、通常はその記述を書いた設計者が自分自身や他人が見て理解しやすいように覚え書きやメモを書くために用いるが、場合によっては記述を削除すること無く無効化するためにも用いる。コメントの形式は 2 種類ある。一つは // から行末までをコメントとするもので、もう一つは /\* で始まって \*/ で終わるまでをコメントとするものである。

#### コメントの例

```
assign x = in; // x に in を代入する           // 以降がコメント
/* this line is a comment
this line is also a comment */                複数行のコメント
```

### A.3 モジュール記述

Verilog-HDL では「モジュール」を基本単位としてハードウェアを記述する。意味のある Verilog-HDL 記述は最低でも一つのモジュールを含む。モジュールの基本的な構造を以下に示す。

モジュールの構造

```
module <モジュール名>(<ポート名>, <ポート名>, ...);
    <ポート宣言>
    <パラメータ宣言>
    <wire 宣言>
    <reg 宣言>

    <本体>
endmodule
```

モジュール名およびポート名は前述の識別子を用いる。ここでのポート名の順序とは無関係な順序で以下のポート宣言を行っても良い。Verilog-HDL では通常は文の終りは ; (セミコロン) で終わるが、endXXXX 系の予約語の文はその一語で文が終わることが明らかなのでセミコロンをつけない。この場合、endmodule 文がこれに相当する。逆に module 文の最後にはセミコロンが必要となるので注意すること。

ポート宣言は以下の構造を持つ。

ポート宣言

```
input  <範囲指定> <信号名>, <信号名>, ...; または
output <範囲指定> <信号名>, <信号名>, ...; または
inout  <範囲指定> <信号名>, <信号名>, ...;
```

input は入力用のポート宣言、output は出力用のポート宣言、inout は入出力用のポート宣言となっている。範囲指定は [ $\text{<MSB>}$  :  $\text{<LSB>}$ ] の形式でその信号線の最上位ビット (MSB) の位置と最下位ビット (LSB) の位置を指定する。範囲指定が省略された場合には  $\text{MSB} = \text{LSB} = 0$  と見なす。

wire 宣言および reg 宣言はモジュール内部で用いられる信号線の宣言を行う。形式はどちらも共通で、以下のような構造を持つ。

wire 宣言, reg 宣言

```
wire  <範囲指定> <信号名>, <信号名>, ...;
reg   <範囲指定> <信号名>, <信号名>, ...;
```

注意が必要なのは、内部で reg 宣言した信号を外部出力に用いるときに、reg 宣言と同時に output 宣言もなければならぬということである。ただし、後述のように Verilog2001 の形式でモジュール宣言を行えばこの記述は一つにまとめることができる。また、モジュール内部で用いられているのに wire 宣言も reg 宣言もされていない信号はエラーにはならず、1 ビット ([0:0] という範囲指定) の wire と見なされるので、記述ミスから信号名を間違ってもコンパイルエラーとならずに別のバグを引き起こすおそれがある。

パラメータ宣言は以下の構造を持つ。

## パラメータ宣言

```
parameter <識別子> = <数値>;
```

## 使用例

```
parameter MEMSIZE = 256;
parameter BITWIDTH = 8;
.
.
.
reg [7:0]          mem[0:MEMSIZE - 1];
wire [BITWIDTH:0] syncro;
```

モジュールの本体はさまざまな記述が現れるが、合成でもっとも良く用いるのは、assign 文と always 文 およびインスタンス記述である。

## A.3.1 Verilog2001 のモジュール宣言

Verilog2001 で上記に加えて以下の形式でモジュールを宣言できるようになっている。

## Verilog2001 のモジュールの構造

```
module <モジュール名>(<ポート宣言>, <ポート宣言>, ...);
    <パラメータ宣言>
    <wire 宣言>
    <reg 宣言>

    <本体>
endmodule
```

オリジナルの Verilog ではモジュール名の後のカッコ内でポート名だけを宣言しておいて、ポート自体の宣言はモジュール内部で行っていたが、Verilog2001 ではモジュール名の後のカッコ内でポート宣言が行えるようになっている。通常はこの形式のほうがタイプする量が少なくて済むのでこちらの形式を使うことが望ましい。どうじにポート名とポート宣言で識別子名をタイプミスで異なる名前を使ってしまうエラーも防ぐことができる。厳密にはパラメータ宣言もモジュール名の後のカッコ内に含めることができるがここでは省略する。

## A.4 assign 文

assign 文では常になり立っている値の代入を表す。

## assign 文

```
assign <信号線名> = <式>;
```

通常は左辺の信号線名は wire 宣言された信号線名そのものであるが、場合によっては以下のような範囲指定を伴ったものや連結演算子を用いたものもある。

## assign 文の例

```
wire [15:0] bus;
wire [3:0] a;
wire [3:0] b;
wire      carry;
wire [3:0] sum;

assign bus[15:12] = a;          (1)
assign {carry, sum} = a + b;    (2)
```

(1) では bus という信号線の 12 ビット目から 15 ビット目までの 4 ビットに 4 ビットの信号線 a を接続している。このような bus[15:12] という表記は範囲指定と呼ばれる。本来は 16 ビットの幅を持つ信号線 bus の一部のみを対象にするとときに用いられる。範囲指定は式の右辺で用いることも可能である。また、bus[15] のように範囲を示す数が 1 つの時には 1 ビットの信号線を表す。この例では bus[15:15] と等価となる。

(2) では 4 ビット同士の加算 (答えは 5 ビットとなる) の結果の上位 1 ビット (5 ビット目) を carry に、下位 4 ビットを sum に接続している。連結演算については次節を参照のこと。

assign 文で注意が必要なのは左辺で現れる信号線は wire 宣言されたものである、ということである。reg 宣言された信号線を assign 文の左辺に持ってくることはできない。また、assign 文は後述する always 文の中のブロックにおくことはできない。常に module ブロックの直下に記述すること。

assign 文の右辺には後述するような演算子と function 文から構成された式を書くことができる。論理合成のための記述として捉えると、assign 文の右辺には「組合わせ回路」を生成するための記述が書かれている、と見なすことができる。

## A.5 演算子

Verilog-HDL には表 A.2 に示すような演算子が定義されている (一部、論理合成で用いない演算子を省略している)。

表 A.3 に演算子の優先順位を示す。同じ優先順位を持つ演算子が並んでいた場合には式の左側に現れる演算子が優先される (表の並び方ではなく、Verilog-HDL 記述の中の式での並び方である)。ただし、この優先順位表にしたがって暗黙の内に優先順位をしているよりも明示的に括弧 ( ) を使ったほうが誤りが少なく、また他人が読んで理解しやすい。また、「Verilog-HDL 論理回路設計」[4] の 22 ページの表 1.3 はビットワイズ演算とリダクション演算を混同しており誤っているので注意が必要である。

### A.5.1 算術演算

通常のプログラミング言語と同様に加算、減算、除算、乗算、剰余の演算子が用意されている。Verilog-HDL では常に符号無し整数として演算が行われる。このうち (△) がついた演算子、乗算、除算、剰余は論理合成すると巨大な回路が合成されるので注意が必要なので、本演習では使用しない。

### A.5.2 関係演算

2 つの値が等しいか (==), 等しくないか (!=), または大小関係が成り立っているかを調べる演算で、結果は 1 ビットの値 (1 か 0) となる。もちろん、成り立っているときに 1 となる。

記号	意味	記号	意味	記号	意味
+	加算	-	減算	*	乗算 (△)
/	除算 (△)	%	剰余 (△)		
<	小なり	<=	小なりイコール		
>	大なり	>=	大なりイコール		
==	一致	!=	不一致		
<<	左シフト	>>	右シフト		
!	論理否定	&&	論理積		論理和
~	ビットワイズ NOT	&	ビットワイズ AND		ビットワイズ OR
^	ビットワイズ XOR	^^	ビットワイズ XNOR		
&	リダクション AND	~&	リダクション NAND		
	リダクション OR	~	リダクション NOR		
^	リダクション XOR	^^	リダクション XNOR		
?:	条件演算	{ }	連結演算		

表 A.2 演算子一覧

優先順位	演算子
高い	単項演算の ! & ~&   ~  ^ ^^ + - * / % 二項演算の + - << >> < <= > >= == != 二項演算の & ^ ^^ 二項演算の   &&    低い ?:

表 A.3 演算子の優先順位

### A.5.3 論理演算とビットワイズ演算

形が似ているが論理演算 (! && ||) は 1 ビットの論理値に対する演算であり、ビットワイズ演算 (~ & | ^ ^^) は多ビットの値に対してビットごとに演算を行うものである。

### A.5.4 リダクション演算

リダクション演算は多ビットの信号線 (もしくは多ビットの値を表す項) の先頭につけるもので、そのすべてのビットを入力とした論理演算を表している。演算結果は常に 1 ビットとなる。

リダクション演算の例

```
wire [15:0] bus;
wire [3:0]  a, b;
wire       c, d;

assign c = ~& bus;           (1)
assign d = ^(a + b);        (2)
```

(1) では 15 ビットの信号線 bus のすべてのビットの NAND (AND + NOT) を計算し、それを c に代入している。(2) では 4 ビットの信号線 a と b のビットワイズ OR を計算した後で、その 4 ビットの値の XOR を計算し、d に代入している。

### A.5.5 条件演算子

条件演算子 (?:) は 3 項演算子で、

1. 条件を表す式
2. 条件が成り立ったときに評価される式
3. 条件が成り立たなかったときに評価される式

の 3 つの式を引数としてとる。意味は上の引数の説明の通りで、条件に応じて信号線の値を切り替えるために用いる。

条件演算子の例

```
wire sel;                                // 選択信号
wire [3:0] input_A, input_B;           // 2つのデータ
wire [3:0] output;                     // 出力

// sel が 1 の時 output に input_A を代入 (接続) し、
// sel が 0 の時 output に input_B を代入 (接続) する。
assign output = sel ? input_A : input_B;
```

### A.5.6 連結演算子

連結演算は複数の信号線をまとめて一つの多ビット信号線のように見なすための表記法で、以下のように用いる。



## 連結演算の例 1

```
wire [7:0] opcode;
wire [3:0] opr_a, opr_b;
wire [15:0] word;
wire [15:0] bus;
wire [7:0] addrh, addrl;

assign { opcode, opr_a, opr_b } = word; (1)
assign bus = { addrh, addrl }; (2)
```

(1) は assign 文の左辺に用いた例でこの場合、16 ビットの値 word をそれぞれ 8 ビット、4 ビット、4 ビットの信号線 opcode, opr\_a, opr\_b に代入している。(2) は assign 文の右辺に用いた例で 2 つの 8 ビット信号線 addrh と addl を結合して 16 ビットの信号線 bus に接続している。これらの例では連結演算は assign 文の中で用いられているが、通常の高ビット信号線が書ける場所ならばどこでもこの連結演算を用いることができる。

また、以下のような形で同一パタンの繰り返しを表すことができる。繰り返しを伴う連結演算では { } 2 組用いる。一つめの { の後に繰り返し回数を記述し、2 つめの { } 内に繰り返しのパターンを記述する。

## 連結演算の例 2

```
wire [9:0] pat0;
wire [15:0] word16;
wire [7:0] word8;

assign pat0 = { 5{ 2'b10 } }; (1)
assign word16 = { { 8{ word8[7] } }, word8 }; (2)
```

(1) では pat0 に 10'b1010101010 という値を代入している。(2) は少し複雑だが、word16 の上位 8 ビットに word[8] の 8 ビット目の値をコピーしている。word16 の下位 8 ビットは word8 をそのままコピーしている。これは「符号拡張」と呼ばれるもので、word8 が 2 の補数表現形式の場合に、符号を保ったまま 16 ビットに拡張するときに用いられる。

## A.6 if 文

if 文は以下のような形式を持つ。

if 文

```
if ( <条件式> ) <ブロック 1>
```

もしくは,

```
if ( <条件式> ) <ブロック 1>  
else <ブロック 2>
```

もしくは,

```
if ( <条件式 1> ) <ブロック 1>  
else if ( <条件式 2> ) <ブロック 2>  
else <ブロック 3>
```

条件式には 1 ビットの式を記述する。ブロックは単文もしくは `begin ~ end` で囲まれた複文を記述する。ただし、単文を `begin ~ end` で囲んでも問題ないので、読みやすさや将来の変更で複文になる場合などを考えると常に `begin ~ end` で囲んでおいた方がよい。

if 文は後述の function 文および always 文の内部でしか用いることはできないので注意が必要である。

## A.7 case 文と casex 文

case 文は以下の形式を持つ。

case 文

```
case ( <式> )  
<値 1>: <ブロック 1>  
<値 2>: <ブロック 2>  
.  
.  
.  
default: <デフォルトブロック>  
endcase
```

式の値が‘値 1’の場合に‘ブロック’が実行される (論理合成の観点で言うと実行されるような回路を合成する, という意味)。このブロックは if 文の場合と同様で単文もしくは `begin ~ end` で囲まれた複文である。最後の `default:` というのは特別な場合で、上記のいずれに該当しない場合にはこの‘デフォルトブロック’が実行される。この `default:` の行は省略可能である。ただし、`default:` の行が存在せず、かつ、そのほかの行がすべての場合を尽くしていない場合の合成結果は不定となるので、とりうる値が多い場合には `default:` 行を入れておいたほうがよい。

casex 文も case 文と同様の構文であるが、値の記述中に‘ドントケア値’を表す‘X’を記述できる。

case 文の例

```
case ( in )
  8'b1xxx_xxxx: penc = 3'd7;
  8'b01xx_xxxx: penc = 3'd6;
  8'b001x_xxxx: penc = 3'd5;
  8'b0001_xxxx: penc = 3'd4;
  8'b0000_1xxx: penc = 3'd3;
  8'b0000_01xx: penc = 3'd2;
  8'b0000_001x: penc = 3'd1;
  8'b0000_0001: penc = 3'd0;
  default:      penc = 3'd0;
endcase
```

この例では優先度つきエンコーダー (priority encoder) を記述している。入力 (in) は 8 ビットで、このうちの 7 ビットめが 1 であれば他のビットの値に関わらず、7 を出力し、7 ビットめが 0 で 6 ビットめが 1 であれば他のビットの値に関わらず 6 を出力する。この回路を case 文を用いて記述するためには  $2^8 = 256$  通りのすべてのパターンを記述しなければならず効率が悪い。

case 文および case 文も後述の function 文および always 文の内部でしか用いることはできないので注意が必要である。

## A.8 function 文

今まで述べた演算子でさまざまな式を記述することができるが、論理演算や算術演算のみではうまく記述できない組み合わせ回路は多数、存在する。そのような場合には function 文を用いて回路の記述を行う。

## function 文の例 1

```
wire [1:0] in;
wire [3:0] out;

function [3:0] decoder;
input [1:0] f_in;
begin
    case (f_in)
        0: decoder = 4'b0001;
        1: decoder = 4'b0010;
        2: decoder = 4'b0100;
        3: decoder = 4'b1000;
    endcase
end
endfunction

assign out = decoder(in);
```

この例では 4 ビットの出力のうち、入力 2 ビットの信号線を 2 進数と見なした場合に対応する数字 (たとえば 2'b10 なら 2) のビットのみを 1 とする回路 (デコーダ) を表している。たとえば入力が 2'b10 なら 2 ビット目のみが 1 のパターン (4'b0100) を出力する。

function 文ではまずその関数の出力の範囲指定を行い、次に関数名 (この例では decoder) を記述する。その次にこの関数に対する入力の宣言を行う。関数は複数の入力をとることができるが、その場合はこの input 文が現れた順番に呼び出し元の信号線との対応がとられることになる。また、function 内部で用いる変数がある場合には reg 宣言を用いて定義を行う。function 文の外側で定義された信号線は一見、function 内部で用いることができるように見えるが、実際には誤りのもととなるので、function の外部の信号線を function 内部で用いるときには明確に function の引数として記述すること。

function 文の内部では assign 文や always 文は用いることができない。代入は=を用いる。その他、用いることができる文は if 文および case 文、casex 文である。function 文を実行した時に返される関数値は function 文で定義された関数名と同名の変数の値が用いられる。この変数は特殊で宣言をする必要がない (というか宣言するとエラーになる)。

function 内部では処理は逐次的に実行されるものと見なされる。たとえば、

## function 文の例 2

```
wire [3:0] x;
wire [1:0] y;
wire [3:0] d;

function [3:0] f;
input [3:0] a;
input [1:0] b;
reg [3:0] c;
begin
    c = a;
    if ( b == 2'b01 ) begin
        c = c + 1;
    end
    f = c << 1;
end
endfunction

assign d = f(x, y);
```

という function 記述があった場合,  $y$  の値が  $2'b01$  で無ければ  $f$  は  $x * 2$  を返し,  $y$  の値が  $2'b01$  なら  $f$  は  $(x + 1) * 2$  を返すことになる.  $f$  という変数は宣言されていないが, 関数名が  $f$  なので自動的に宣言されていることに注意.

## A.8.1 Verilog2001 の function 文

Verilog2001 では以下の形式でも function を定義できる.

function 文の例 2

```
wire [1:0] in;
wire [3:0] out;

function [3:0] decoder(input [1:0] f_in);
begin
    case (f_in)
        0: decoder = 4'b0001;
        1: decoder = 4'b0010;
        2: decoder = 4'b0100;
        3: decoder = 4'b1000;
    endcase
end
endfunction

assign out = decoder(in);
```

モジュール文と同様に入力の変数が関数名の後のカッコ内に入っている。こちらの形式のほうがわかりやすい。

## A.9 always 文

always 文はさまざまな目的で用いられるが、ここでは同期式順序回路を記述するための always のみを解説する。本演習ではこの記述に現れるパタンのみを用いること。

### A.9.1 D-FlipFlop の記述

以下の例は D-FlipFlop の記述である。

D-FlipFlop

```
module dff(clk, d, q);
input clk, d;
output q;

reg q;

always @(posedge clk)
begin
    q <= d;
end

endmodule
```

always 文は '@' の後にこの always ブロックが実行されるきっかけとなるイベントを記述する。この例では

‘posedge clk’ がこのイベントに相当し、clk という信号線の値が 0 から 1 に立ち上がる時、を意味する。そのあとに本体のブロックを記述する。これは if 文や case 文のブロックと等しく、単文か begin ~ end で囲まれた複文であるが、誤りの防ぐためにも常に begin ~ end で囲んでおいた方がよい。

always ブロックの内部では if 文、case 文、casex 文 を用いることはできるが、assign 文や function 文 (function の定義) は記述できない。つまり、assign や function の定義は常に存在しているものであり、あるイベントの度に行われるものではないからである。後述のインスタンス記述も always ブロックの中には記述できない。また、always ブロック内部での代入は ‘<=’ という記法を用いる。この形式の代入はノンブロッキング代入と呼ばれる。上の例では clk が 0 から 1 に変化する度に always ブロックが実行され、入力 d の値が出力 q に代入されている。また、ノンブロッキング代入の左辺の信号線は reg 宣言されていなければならない。上の例では出力 q は wire 宣言も reg 宣言も省略すると暗黙の内に wire 宣言されたものと見なされるので明示的に reg 宣言している。

次の例は 2 つの D-FlipFlop を直列に繋いだものの記述である。

D-FlipFlop の直列接続

```
module dff2(clk, in, out);
input clk, in;
output out;

reg q1;
reg out;

always @(posedge clk)
begin
    q1 <= in;
    out <= q1;
end

endmodule
```

この記述から合成される回路図を図 A.1 に示す。

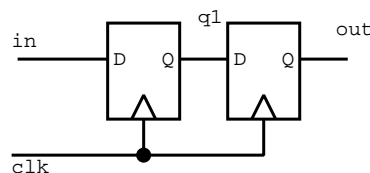


図 A.1 D-FlipFlop の直列接続

プログラミング言語における代入文は (ノンブロッキング代入と区別する意味で) ブロッキング代入と呼ばれる。つまり、一つの代入文の処理が終わらないと次の文には実行が移らない (ブロックしている)。これに対し、ノンブロッキング代入は代入処理の完了を待たずに次の文に実行が移るので、ある意味、すべての文が同時に実行されていると見なすこともできる。上の例で、q1 <= in が行われてから逐次的に out <= q1 が行われるわけではなく、q1 <= in と out <= q1 が同時に行われる。つまり、out には 1 クロック前の q1 の値、

つまり、2 クロック前の in の値が入っている。もしも逐次的に代入が行われるとすると `out <= in` と等価になり、前の D-FlipFlop の記述とこの記述が全く変わらないことになる。ノンブロッキング代入はその記述中に現れる順序に意味はない、そのため、この always ブロック中の代入文の順序が、

```
out <= q1;
q1 <= in;
```

であっても全く同一の回路が合成される。

文法的には 1 つのモジュール内に複数の always ブロックを持つことはできるが、論理合成可能な記述としては、複数の D-FlipFlop があっても、1 つの module 記述中には 1 つの always 文しか作らないようにすること。

### A.9.2 非同期リセットつき D-FlipFlop の記述

次の例は非同期リセットつき D-FlipFlop の記述である。

```
D-FlipFlop
module dff(clk, rst, d, q);
input clk, rst, d;
output q;

reg q;

always @(posedge clk or negedge rst)
begin
    if ( !rst )
        q <= 0;
    else
        q <= d;
end

endmodule
```

この記述の意味は always ブロックが起動されるタイミングが clk の立ち上がりもしくは rst の立ち下がり (1 から 0 への遷移) であり、もしも rst が 0 ならば q に 0 を代入し、そうでなければ (リセット無しの D-FlipFlop と同様に) q に d を代入する、というものである。

以降、順序回路の記述を行う際には、この

```
always @(posedge clk or negedge rst)
begin
    if ( !rst ) begin
        リセット時の初期化代入
    end
    else begin
        通常の動作
    end
end
```



```
end  
end
```

という形式で行うこととする。特に if ( !rst ) begin ~ end のあとの else を忘れると合成できないので注意すること。

この非同期リセットつき D-FlipFlop を 2 個直列接続した回路記述を以下に示す。

非同期リセットつき D-FlipFlop の直列接続

```
module dff2(clk, rst, in, out);  
  input clk, rst, in;  
  output out;  
  
  reg q1;  
  reg out;  
  
  always @(posedge clk or negedge rst)  
  begin  
    if ( !rst ) begin  
      q1 <= 0;  
      out <= 0;  
    end  
    else begin  
      q1 <= in;  
      out <= q1;  
    end  
  end  
end  
  
endmodule
```

リセット無しの記述と異なる点は always の '@' 以降に negedge rst が加わっていることと、if ( !rst ) の部分が加わったことである。

### A.9.3 組合わせ回路を含んだ順序回路記述

もちろん、ただ単に値を代入する以外の複雑な演算を伴った順序回路も記述可能である。以下の例は非同期リセットつき 4 ビット 2 進カウンタの記述である。

非同期リセットつき 4 ビット 2 進カウンタ

```
module count4(clk, rst, q);
input      clk, rst;
output [3:0] q;

reg [3:0] q;

always @(posedge clk or negedge rst)
begin
    if ( !rst )
    begin
        q <= 4'b0000;
    end
    else
    begin
        q <= q + 1;
    end
end
endmodule
```

次の例はこの記述を修正した 10 進カウンタの記述である (この例ではわざと begin~end を極力省いている。どのような時に必要なのかを確認すること).

非同期リセットつき 10 進カウンタ

```
module count10(clk, rst, q);
input          clk, rst;
output [3:0] q;

reg [3:0] q;

// 10 進カウンタの次の値を計算する.
function [3:0] next;
input [3:0];

begin
    next = in + 1;
    if ( next == 10 )
        next = 0;
end
endfunction

always @(posedge clk or negedge rst)
    if ( !rst )
        q <= 4'b0000;
    else
        q <= next(q);
endmodule
```

## A.10 インスタンス記述

今まで説明した構文要素を用いれば基本的な組合わせ回路、順序回路を一つのモジュールとして記述することができるが、大規模な回路を一つのモジュールで記述するのは効率的ではないし、誤りが入りやすくなる。Verilog-HDL では設計済のモジュールを部品として用いてさらに大きなモジュールを構築するための枠組みを用意している。これは回路図エディタ上で設計した回路をシンボル化して他の回路図から用いるのと同様の概念である。Verilog-HDL ではこのようにモジュールを部品として用いることをインスタンス化と呼ぶ。次の例は全加算器 (full adder) を部品として 4 ビット加算器を記述したものである。この例では説明の都合で下位モジュール (fulladder) の信号線と上位モジュール (adder4) の信号線を区別するために大文字と小文字を使って記述しているが、一般にはこのように大文字か小文字かの区別だけで信号線を区別することは好ましくない。

インスタンス記述の例

```

module fulladder(A, B, CIN, S, COUT);
input A, B, CIN;
output S, COUT;

assign S = A ^ B ^ CIN;
assign COUT = A & B | A & CIN | B & CIN;
endmodule

module adder4(a, b, cin, s, cout);
input [3:0] a, b;
input cin;
output [3:0] s;
output cout;

wire tmp1, tmp2, tmp3;

fulladder fa1(a[0], b[0], cin, s[0], tmp1);
fulladder fa2(a[1], b[1], tmp1, s[1], tmp2);
fulladder fa3(a[2], b[2], tmp2, s[2], tmp3);
fulladder fa4(a[3], b[3], tmp3, s[3], cout);

endmodule

```

この記述に対応した回路図を図 A.2 に示す。

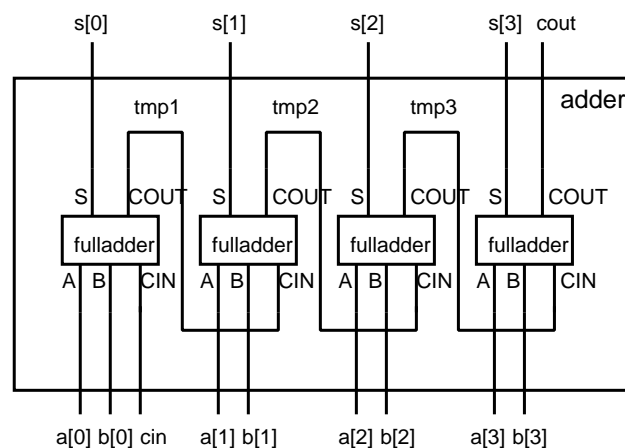


図 A.2 4 ビット加算器

インスタンス化記述は、

<モジュール名> <インスタンス名>(信号線名 1, 信号線名 2, ...)

という形で記述する。ここで‘モジュール名’はインスタンス化する元の下位モジュールの名前 (図 A.2 の例では `fulladder`) である。‘インスタンス名’は個々のインスタンスを区別するための名前である。この例では 4 つの `fulladder` のインスタンスが生成されているのでそれらを区別するためにそれぞれ `fa1`, `fa2`, `fa3`, `fa4` という名前をつけている。

この記述中の `wire tmp1`, `tmp2`, `tmp3` の宣言は実は必要ではなく、このような 1 ビットの `wire` 型信号線は暗黙の内に宣言される。しかし、このような暗黙の宣言にまかせると、

```
fulladder fa2(a[1], b[1], tmp1, s[1], tmp2);
```

のように `tmp1` を `temp1` に書き間違えてもコンパイラは信号線名が未定義というエラーは報告しないのでバグの発見が遅れる可能性がある。このような記述ミスは本来はコンパイラが機械的にチェックするべきであるが、Verilog-HDL を使うときには記述者が注意深くチェックする必要がある。

インスタンス化されたモジュールの入出力ポートに割り当てられる信号線はこの記述例の様に順序によって対応をとる場合と次の例のように明示的にポート名を指定して対応をとる場合がある。後者のほうが記述量が多く煩わしい書き方ではあるが、ポートとの対応づけに関するミスを減らすことができる。

ポート名を指定したインスタンス記述の例

```
fulladder fa1(.A(a[0]), .B(b[0]), .S(s[0]), .CIN(cin), .COUT(tmp1));
```

この形式では

.<下位モジュールのポート名>(<ポートに接続する信号線名>)

の形で接続を記述する。

いずれにせよ、下位モジュールのインスタンス化記述は回路図を用いた回路記述と同程度のものであり、あまりハードウェア記述言語の特徴を活かした可読性のよいものとは言えない。インスタンス記述を用いるときは手元で簡単な回路図 (ブロック図) を書いておいてそれを Verilog-HDL で記述するとか、ポート名と信号線の割り当てが間違っていないかなどのチェックを入念に行う、などの工夫が必要である (本来はハードウェア記述言語がもう少し賢くあるべきではあるが)。また、下位モジュールが簡単な記述の場合には、下位モジュールをインスタンス化するよりも直接上位モジュールにその内容を記述してしまった方が分かりやすいこともある。

インスタンス記述でもうひとつ重要なことは、下位モジュールの出力ポートに接続する信号線は `wire` 型でなければならないということである。これは下位モジュールの出力ポートとの接続が一種の `assign` 文であり、上位モジュールの信号線が `assign` 文の左辺に相当するものだと考えれば当然のことである。入力ポートに接続する信号線はいわゆる `assign` 文の右辺なのでこちらは `wire` 型でも `reg` 型でも構わない\*1。

## A.11 Verilog-HDL 記述の注意点

最後に、Verilog-HDL 記述を行ううえでの注意点をもう一度あげておく。

- 数値にはビット幅の指定をしておいた方がよい。
- 信号線には `wire` 型と `reg` 型がある。

\*1 実際には、単一の `wire` や `reg` でなくて式でよいが、慣れないうちは誤りの元となるのでやらない方がよい。

- assign 文の左辺には wire 型, 右辺はどちらでもよい.
- ノンブロッキング代入 ( $\leq$ ) の左辺は reg 型.
- function 文の内部の変数は reg 型, ただし代入には  $=$  を用いる.
- assign 文, インスタンス記述は always 文の中には書けない.
- if 文, case 文, casex 文は always 文と function 文の中にしか書けない.
- 一つのモジュールには一つの always @(posedge clk or negedge rst).
- インスタンス記述の出力ポートに接続できる信号線は wire 型.
- タイプミスなどで宣言していない信号線を記述しても暗黙の内に 1 ビットの wire 型信号線とみなされるので目視チェックをよく行うこと.
- begin ~ end や if ~ else の対応に気をつけること.
- Quartus を用いて論理合成する場合のみのルール:  
プロジェクト名は最上位のモジュール名と同一名とすること. また, 一つのモジュールを一つのファイルに記述し, ファイル名を 'モジュール名'.v としておくこと.
- ルールではないが守ったほうがよいガイドライン:
  - 1 つのファイルに 1 つのモジュールを記述する.
  - そのファイル名は 'モジュール名'.v とする.
- その他, おかしな現象に出会ったときには教員や TA に尋ねること.

## 付録 B

# FPGA ボードの仕様

ここでは実験に用いる FPGA ボード MU500(MU500-RX, MU500-RK, MU500-7SEG) の仕様について述べる。

### B.1 概要

MU500-RX は ALTERA(Intel) 社製の FPGA である CycloneIV を搭載した FPGA ボード\*1である。上下のコネクタにより、キースイッチ等を搭載したユーザーインターフェイスボード MU500-RK および多数の 7SEG-LED を搭載した MU500-7SEG と接続されている。

図 B.1 に MU500-RX, MU500-RK, MU500-7SEG ボードの各部名称を示す。

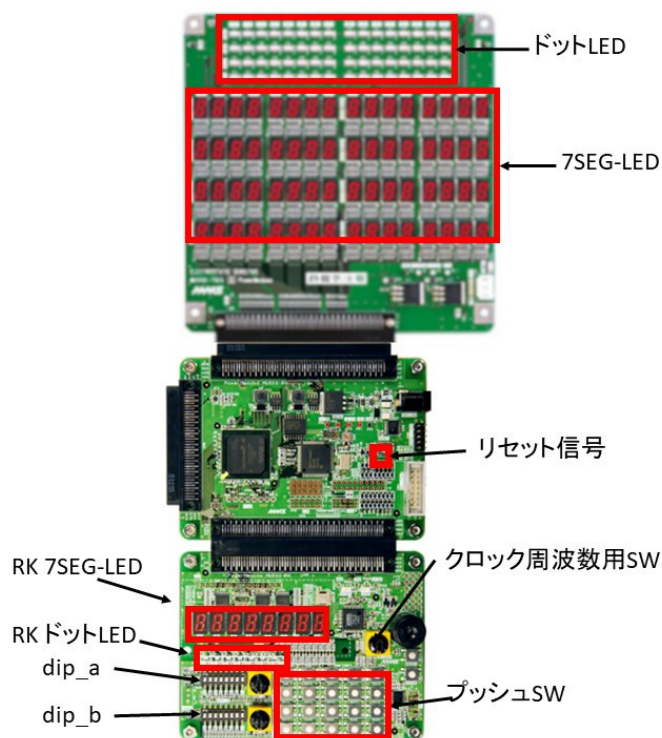


図 B.1 FPGA ボードの各部名称

\*1 他の Renesas 社製のマイコン RX210 も搭載しているが本実験では使用しない

中央の MU500-RX ボード上で使用するスイッチは右側中央のリセットスイッチ (白地に RESET と刻印されている) のみである (図 B.1 のリセット信号)。また、電源プラグの近くに通電状態を示す LED がある。

下部の MU500-RK ボードは以下のインターフェイスを持つ。

- 20 個のテンキースイッチ
- 3 つのロータリスイッチ
- 8 つの 7 セグメント LED
- 8 つのドット LED

このうち右側中央のロータリースイッチ (図 B.1 のクロック周波数用 SW) はクロック周波数を設定する目的で使用される。スイッチの値と周波数の対応を表 B.1 に示す。

表 B.1 CK\_DIV とクロック周波数の対応

CK_DIV	0	1	2	3	4	5	6	7
周波数	40MHz	20MHz	10MHz	5MHz	1.25MHz	312.5kHz	78.1kHz	9.8kHz

CK_DIV	8	9	A	B	C	D	E	F
周波数	9.8kHz	4.9kHz	2.44kHz	1.22kHz	610Hz	305Hz	1.0Hz	*

CK\_DIV = F の場合 (表中の \*) は特殊で、ボード右端の CLKSW のプッシュボタンを押した時に 1 つ分のパルスがクロック信号に印加される。

それ以外のスイッチおよび LED はそのまま FPGA に接続されており、FPGA 側で適切な回路を用意することで自由に使用することができる。本実験ではすべての課題において同一のピン割り当てを用いるため、Verilog-HDL でトップレベルのモジュールを定義する時には以下の識別子名を用いること。



表 B.2 MU500-RX, MU500-RK 用の入出力宣言

Verilog-HDL 記述	説明
<code>input sys_clock</code>	システム全体のクロック。周波数は 20MHz に固定されている。
<code>input reset</code>	リセット信号。RESET ボタンに直結する。 負論理 (スイッチを押した時に 0 となる) であることに注意。
<code>input clock</code>	クロック周波数用 SW で周波数を変えることのできるクロック信号。
<code>input psw_a0, —, psw_a4</code>	テンキーの 1 行目 (横方向) の 5 つのスイッチ。 負論理 (スイッチを押した時に 0 となる) であることに注意。
<code>input psw_b0, —, psw_b4</code>	テンキーの 2 行目 (横方向) の 5 つのスイッチ。 負論理 (スイッチを押した時に 0 となる) であることに注意。
<code>input psw_c0, —, psw_c4</code>	テンキーの 3 行目 (横方向) の 5 つのスイッチ。 負論理 (スイッチを押した時に 0 となる) であることに注意。
<code>input psw_d0, —, psw_d4</code>	テンキーの 4 行目 (横方向) の 5 つのスイッチ。 負論理 (スイッチを押した時に 0 となる) であることに注意。
<code>input [3:0] ]hex_a</code>	2 つ並んだロータリースイッチの上側のスイッチの出力。 0 – F(15) の値を 4 ビットの信号線で表す。
<code>input [3:0] ]hex_b</code>	2 つ並んだロータリースイッチの下側のスイッチの出力。 0 – F(15) の値を 4 ビットの信号線で表す。
<code>input [7:0] dip_a</code>	2 つ並んだ DIP スイッチの上側のスイッチの出力。 8 つのスイッチの値を 8 ビットの信号線で表す。 スイッチが上側にある時に 1 となる。
<code>input [7:0] dip_b</code>	2 つ並んだ DIP スイッチの下側のスイッチの出力。 8 つのスイッチの値を 8 ビットの信号線で表す。 スイッチが上側にある時に 1 となる。
<code>output [7:0] seg_x</code> <code>output [3:0] sel_x</code>	RK ボード上の 8 つの 7SEG-LED のうち左半分の 4 つ分 (A, B, C, D) に対する制御信号。詳細は後述。
<code>output [7:0] seg_y</code> <code>output [3:0] sel_y</code>	RK ボード上の 8 つの 7SEG-LED のうち右半分の 4 つ分 (E, F, G, H) に対する制御信号。詳細は後述。
<code>output [7:0] led_out</code>	8 つ並んだドット LED に接続する 8 ビットの信号線。

図 B.2 にプッシュ SW の信号線名の対応を示す。なお、このプッシュ SW は負論理となっていて、通常は 1 であるが、押された時だけ 0 となることに注意。

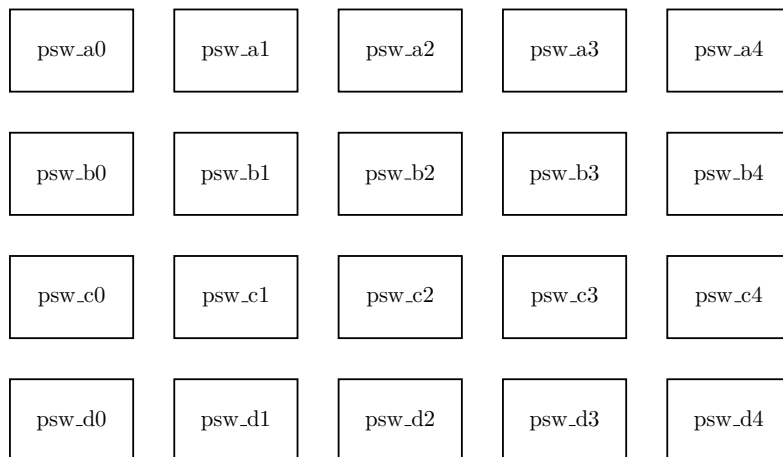


図 B.2 プッシュ SW のレイアウト

MU500-RX と MU500-7SEG はそれぞれに 7SEG-LED とドット LED を持つ。以降ではそれらを区別するために MU500-RK に搭載されているものをそれぞれ RK 7SEG-LED, RK ドット LED と呼ぶことにする。MU500-7SEG に搭載されているものはそのまま 7SEG-LED, ドット LED と呼ぶ。

7SEG-LED は実際には 7 つのセグメントと 1 つのドット (小数点の位置にある) があるので 1 つ分の制御に 8 ビット必要となる。8 個分の 7SEG-LED を同時に制御する場合には単純には  $8 \times 8 = 64$  ビットの信号線が必要となる。ところが FPGA のピン数は数百~千のオーダーであり、8 個の 7SEG-LED のために 64 個のピンを専有してしまうのは効率が悪い。そこで、MU500-RK では 4 つの 7SEG-LED を 1 グループとして 4 つのセグメント信号を共有し (`seg_x` と `seg_y`)、セレクト信号 (`sel_x` と `sel_y`) で現在のセグメント信号がどの 7SEG-LED をドライブしているかを示すようにしている。つまり、4 つの 7SEG-LED のうち同時には 1 つしかドライブできない。そこで、7SEG-LED を表示させるには時分割で出力信号を制御する回路が必要となる。

図 B.3 に各セグメントのビット割り当てを示す。

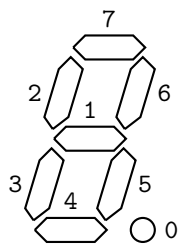


図 B.3 7セグメント LED のビット割り当て

表 B.3 に数字を表すパターンの例を示す。

表 B.3 7セグメント LED の数字パターンの例

a[7:0]		a[7:0]		a[7:0]		a[7:0]	
1111_1100	0	0110_0110	4	1111_1110	8	0001_1010	2
0110_0000	1	1011_0110	5	1111_0110	9	0111_1010	3
1101_1010	2	1011_1110	6	1110_1110	8	1001_1110	6
1111_0010	3	1110_0000	7	0011_1110	6	1000_1110	7

図 B.4 に RK 7SEG-LED のレイアウトを示す。

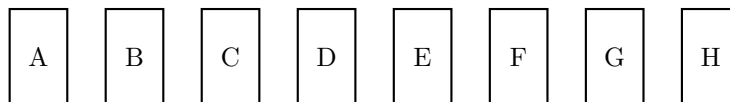


図 B.4 RK 7SEG-LED のレイアウト

RK 7SEG-LED の真下にある 8 つの RK ドット LED も RK 7SEG-LED と同様のレイアウトとなっているが、トップレベルのインターフェイスとしては 8 ビットの信号線 `led_out` にまとめている。各ビットと LED の対応は表 B.4 の様になっている。

表 B.4 RK ドット LED のレイアウト

A	B	C	D	E	F	G	H
<code>led_out[0]</code>	<code>led_out[1]</code>	<code>led_out[2]</code>	<code>led_out[3]</code>	<code>led_out[4]</code>	<code>led_out[5]</code>	<code>led_out[6]</code>	<code>led_out[7]</code>

次に表示用のボードである MU500-7SEG について説明する。MU500-7SEG は 64 個の 7SEG-LED と 64 個のドット LED からなる。FPGA からみると全て出力となっている。以下に Verilog-HDL 記述で用いる識別子名を示す。

表 B.5 MU500-7SEG 用の出力宣言

Verilog-HDL 記述	説明
input [7:0] seg_a	A グループ (1 行目の左半分) のセグメント信号
input [7:0] seg_b	B グループ (1 行目の右半分) のセグメント信号
input [7:0] seg_c	C グループ (2 行目の左半分) のセグメント信号
input [7:0] seg_d	D グループ (2 行目の右半分) のセグメント信号
input [7:0] seg_e	E グループ (3 行目の左半分) のセグメント信号
input [7:0] seg_f	F グループ (3 行目の右半分) のセグメント信号
input [7:0] seg_g	G グループ (4 行目の左半分) のセグメント信号
input [7:0] seg_h	H グループ (4 行目の右半分) のセグメント信号
input [8:0] sel	選択信号

仕組みは MU500-RK の 7SEG-LED とほぼ同様で、こちらは 8 個の 7SEG-LED を 1 つのグループにしている。ただし、さらに 64 個のドット LED も同様のグループ分けを行っており、各グループは 8 個の 7SEG-LED と 8 ドット分の LED の計 9 種類のセグメント信号を共有している。そのため選択信号 `sel` は 9 ビットとなっている。こちらも時分割で表示を行うためにドライブする制御回路が必要となる。

## B.2 MU500-RK, MU500-7SEG 用表示ドライバ

前述の様に MU500-RK および MU500-7SEG の 7SEG-LED は時分割でセグメント信号を共有しているので、順序回路で制御を行う必要がある。本実験ではこの表示ドライバ回路はあらかじめ設計したものを使用する。以下に外部仕様を示す。

MU500-RK, MU500-7SEG 用表示ドライバ回路

```
module led_driver(input      sys_clock, // システムクロック
                  input      reset,    // リセット
                  input [63:0] seg7_a, // MU500-7SEG の A グループ のセグメント信号
                  input [63:0] seg7_b, // MU500-7SEG の B グループ のセグメント信号
                  input [63:0] seg7_c, // MU500-7SEG の C グループ のセグメント信号
                  input [63:0] seg7_d, // MU500-7SEG の D グループ のセグメント信号
                  input [63:0] seg7_e, // MU500-7SEG の E グループ のセグメント信号
                  input [63:0] seg7_f, // MU500-7SEG の F グループ のセグメント信号
                  input [63:0] seg7_g, // MU500-7SEG の G グループ のセグメント信号
                  input [63:0] seg7_h, // MU500-7SEG の H グループ のセグメント信号
                  input [63:0] seg7_dot64, // MU500-7SEG のドット LED の出力信号
                  input [7:0] rk_a,    // MU500-RK の A セグメント信号
                  input [7:0] rk_b,    // MU500-RK の B セグメント信号
                  input [7:0] rk_c,    // MU500-RK の C セグメント信号
                  input [7:0] rk_d,    // MU500-RK の D セグメント信号
                  input [7:0] rk_e,    // MU500-RK の E セグメント信号
                  input [7:0] rk_f,    // MU500-RK の F セグメント信号
                  input [7:0] rk_g,    // MU500-RK の G セグメント信号
                  input [7:0] rk_h,    // MU500-RK の H セグメント信号
                  output [7:0] seg_a,  // MU500-7SEG のボード出力 (seg_a)
                  output [7:0] seg_b,  // MU500-7SEG のボード出力 (seg_b)
                  output [7:0] seg_c,  // MU500-7SEG のボード出力 (seg_c)
                  output [7:0] seg_d,  // MU500-7SEG のボード出力 (seg_d)
                  output [7:0] seg_e,  // MU500-7SEG のボード出力 (seg_e)
                  output [7:0] seg_f,  // MU500-7SEG のボード出力 (seg_f)
                  output [7:0] seg_g,  // MU500-7SEG のボード出力 (seg_g)
                  output [7:0] seg_h,  // MU500-7SEG のボード出力 (seg_h)
                  output [8:0] sel,    // MU500-7SEG のボード出力 (sel)
                  output [7:0] seg_x,  // MU500-RK のボード出力 (seg_x)
                  output [3:0] sel_x,  // MU500-RK のボード出力 (sel_x)
                  output [7:0] seg_y,  // MU500-RK のボード出力 (seg_y)
                  output [3:0] sel_y); // MU500-RK のボード出力 (sel_y)

    ...

endmodule
```

`seg7_a`, `seg7_b`, `seg7_c`, `seg7_d`, `seg7_e`, `seg7_f`, `seg7_g`, `seg7_h` はそれぞれ 8 個の 7SEG-LED に表示する内容を表す信号線である。1 つの 7SEG-LED に 8 ビット必要なので 1 つのグループ (7SEG-LED 8 個) で 64 ビットとなっている。`seg7_dot64` は 64 個のドット LED の出力信号で、左上から右方向に 0, 1, ... とインデックス付けされている。これらの入力を適切に時分割で切り替えて `seg_a`~`seg_h` および `sel` に出力する。`rk_a`, `rk_b`, `rk_c`, `rk_d`, `rk_e`, `rk_f`, `rk_g`, `rk_h` は MU500-RK の 8 つの 7SEG-LED に表示する内容を表す信号線である。これを適切に時分割で切り替えて `seg_x`, `sel_x`, `seg_y`, `sel_y` に出力する。

## 付録 C

# 教育用プロセッサ KAPPA3-RV32I の仕様 Ver. 0.1

### C.1 概要

この文書は教育用プロセッサ KAPPA3-RV32I(Kyushu Advanced Program for Processor Architecture Ver. 3 -RV32I) の仕様について記したものである。ここで述べるプロセッサは、教育用として最低限必要な機能を持ち、かつ学生が理解しやすい構造を採ることを目標とする。ベースとなるアーキテクチャとして設計を容易にし、かつパイプラインによる実装への発展を理解させるため、汎用レジスタを持ったロードストア型アーキテクチャとしてオープンなアーキテクチャ RISC-V(RV32I) を採用している。RISC-V では語長や命令セットの種類によっていくつかのバリエーションを持つが、ここでは 32 ビットの整数演算命令のみをサポートする RV32I を用いる。他の一般的な RISC(Reduced Instruction Set Computer) プロセッサと同様に、全ての算術論理演算はレジスタ・レジスタ間もしくはレジスタ・即値の間で行われる。以下に主な仕様を述べる。

語長:     • 32 ビットを 1 語とする。

- バイトアドレッシング — 1 バイト (8 ビット) 単位でメモリ番地がついている。つまり、1 語は 4 バイトからなる。
- リトルエンディアン — 4 バイトの下位のバイトが先 (下位アドレス) にくる形式のこと。つまり 32'h12345678 (Verilog-HDL の表記) という 32 ビットの値を 0 番地から始まる 4 バイト (つまり 0 番地から 3 番地) に書き込むと 0 番地の値は下位 8 ビットの 8'h78 となり 1 番地の値は次の 8 ビットの 8'h56, 最後の 3 番地は 8'h12 となる。
- 全てのアクセスは整列化されていなければならない。4 バイトのワード (語) に対するアクセスは必ず 4 の倍数のアドレスに対して行われなければならない。2 バイトのハーフワード (半語) に対するアクセスは必ず 2 の倍数のアドレスに対して行われなければならない。

内部メモリ:   FPGA チップ内に 64K バイトの内部メモリを持つ。ただしあらかじめ設計された記述を与えるので学生は設計する必要はない。

タイマー:   プロセッサとは独立に動作する 64 ビットのタイマーカウンタを持つ。タイマーのカウント値が指定された値に一致した時に割り込み要求が発生する。

割り込み:   単純な割り込み機能を持つ。割り込み要求に応じて通常のプログラムの実行を中断して別のプログラムの実行を行う。

KAPPA3-RV32I は RISC-V の RV32I の仕様に準拠したものであるが、そのままでは学生実験の題材として複雑すぎるので、KAPPA3-RV32I から割り込み関係の機能を削除して簡単化したプロセッサである

KAPPA3-LIGHT を用意した．本実験ではまずこの KAPPA3-LIGHT の設計および動作確認を行う．引き続き，KAPPA3-RV32I を用いたソフトウェア開発の演習を行う．以降では KAPPA3-RV32I の仕様について述べるが割り込みと CSR(後述) に関する記述以外は KAPPA3-LIGHT も同様である．

## C.2 命令フォーマット

全ての命令は 32 ビット (1 語) 固定長であり，次の 6 種類の形式を持つ．なお，命令の形式に関わらず最下位 7 ビットはオペコード (opcode) と呼ばれるフィールドで命令の種類を表す．

表 C.1 命令フォーマットの種類

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

**R-type:** 主にレジスタ・レジスタ演算命令に使用される． $R_d$  はデスティネーションレジスタと呼ばれる．演算結果もしくはロードした値を格納するレジスタを指定する． $R_{s1}, R_{s2}$  はソースレジスタと呼ばれる．演算命令の場合にはその名の通りソースとなる値を格納しているレジスタを指定する．残りの 3 ビットの funct3 フィールドと 7 ビットの funct7 フィールドで命令の詳細な機能を指定する．

**I-type:** レジスタ・即値演算，レジスタ間接ジャンプ (JALR)，およびロード命令に使用される． $R_d$  はデスティネーションレジスタと呼ばれる．通常は演算結果を格納するレジスタを指定するが，ジャンプ命令の場合には戻り値 (現在の PC の値) を格納するレジスタの指定に用いる． $R_{s1}$  はソースレジスタと呼ばれる．演算に用いられるレジスタを指定する．ジャンプ命令およびロード命令の場合にはアドレスを表すベースレジスタとして用いられる．上位 12 ビットの即値は演算命令の場合は演算に用いる値として用いられる．ロード命令およびジャンプ命令の場合にはアドレスのオフセットとして用いられる．

**S-type:** ストア命令で用いられる． $R_{s1}$  レジスタはアドレス計算のベースレジスタとして用いられる． $R_{s2}$  レジスタはストアする値を表している．ロード命令と同じく即値は 12 ビットで表されるが 2 つのフィールドにまたがっている．これは全ての命令のなかで  $R_{s1}, R_{s2}, R_d$  レジスタの指定フィールドを同一にするための工夫である．

**B-type:** 分岐命令で用いられる．S-type に似ているが，即値のエンコーディングが異なっている．これは分岐先アドレスが偶数であることから最下位ビット (0 ビット) が不要であることから即値を 1 ビットずらして用いるための工夫である．ただし，即値の 12 ビットと 11 ビット以外は S-type と同一のエンコーディングになっている．

**U-type:** 上位ロード命令 (LUI, AUIPC) で用いられる． $R_d$  フィールドとオペコード以外の 20 ビットを即値として用いている．ただし，31 ビット目から 12 ビット目までの上位 20 ビットを表していることに注意．

**J-type:** ジャンプ命令で用いられる．こちらも U-type と同様に  $R_d$  フィールドとオペコード以外の 20 ビットを即値として用いているがエンコーディングが異なる．こちらは即値の 10 ビット目から 1 ビット目までが I-type と同一になるように工夫されている．なお，ジャンプ先のアドレスは偶数なので最下位



ビットは常に 0 となるので指定しない。

## C.3 アドレッシングモード

アドレッシングモードとはメモリ上の位置 (メモリ番地) を指定する方法のことである。KAPPA3-RV32I では大まかには 1 種類のアドレッシングモードしかサポートしない。これは ‘指定されたレジスタ (ベースレジスタ) の値’ + ‘オフセット’ の形で与えられる。ただしベースレジスタの種類とオフセットの形式で以下の 4 種類がある。

- I-type:  $R_{s1}$  がベースレジスタとして用いられる。12 ビットの即値は符号拡張されてベースレジスタの値と加算される。
- S-type: I-type と同じく  $R_{s1}$  がベースレジスタとして用いられ、12 ビットの即値は符号拡張されてベースレジスタの値と加算される。ただし、即値のフィールドが I-type と異なっている。
- B-type: 命令中には明示されていないが、PC (プログラムカウンタ) がベースレジスタとして用いられる。さらに 12 ビットの即値は符号拡張されたあとで 2 倍してから PC の値に加算される。
- J-type: こちらも PC をベースレジスタとして使用する。J-type の即値は 20 ビットであるが、S-type と同様に符号拡張されたあとで 2 倍してから PC に加算される。

このようにロード・ストア命令では汎用レジスタをベースレジスタに用い、分岐・ジャンプ命令では PC をベースレジスタに用いるようになっている。ただし、JALR 命令だけは例外で  $R_{s1}$  フィールドで指定された汎用レジスタがベースレジスタとして用いられる。この命令のみが現在の PC と無関係なアドレスにジャンプすることができる。他の分岐・ジャンプ命令が PC 相対アドレスを用いている理由は、プログラムがどこに配置されても命令中の分岐先アドレスの指定を書き換える必要がないようにするためである。このようにプログラムの配置位置によって内容を書き換える必要のないコードを PIC (position independent code) コードと呼ぶ。PIC コードを用いることでプログラム開発で用いられるリンカ・ローダの処理が大幅に簡単化される。

## C.4 命令セット

### C.4.1 即値ロードとジャンプ命令

表 C.2 命令セット (1)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
imm[31:12]										rd		0110111		LUI
imm[31:12]										rd		0010111		AUIPC
imm[20 10:1 11 19:12]										rd		1101111		JAL
imm[11:0]						rs1		000		rd		1100111		JALR
imm[12 10:5]				rs2		rs1		000		imm[4:1 11]		1100011		BEQ
imm[12 10:5]				rs2		rs1		001		imm[4:1 11]		1100011		BNE
imm[12 10:5]				rs2		rs1		100		imm[4:1 11]		1100011		BLT
imm[12 10:5]				rs2		rs1		101		imm[4:1 11]		1100011		BGE
imm[12 10:5]				rs2		rs1		110		imm[4:1 11]		1100011		BLTU
imm[12 10:5]				rs2		rs1		111		imm[4:1 11]		1100011		BGEU

LUI(Load Upper Immediate) : U-Type

レジスタの上位 20 ビットに即値をロードする。下位 12 ビットは常に 0 となる。

AUIPC(Add Upper Immediate to PC) : U-type

PC に即値を足す。ただし、即値は上位 20 ビットを指定したもの。下位 12 ビットは 0 を足すものとみなす。

JAL(Jump And Link) : J-type

次の PC の値 (自分自身のアドレス +4) を  $R_d$  に入れ、指定されたアドレスにジャンプする。 $R_d$  は戻り先のアドレスとして用いられる。

JALR(Jump And Link Register) : I-type

次の PC の値 (自分自身のアドレス +4) を  $R_d$  に入れ、 $R_{s1} + imm$  のアドレスにジャンプする。 $R_d$  は戻り先のアドレスとして用いられる。

ここで  $imm$  は即値フィールドで指定された即値である。 $imm$  は符号付き 12 ビット整数として扱われる。

BEQ(Branch Equal) : B-type

$R_{s1} = R_{s2}$  の時に現在の PC の値 (自分自身のアドレス) に即値を加えたアドレスにジャンプする。即値は符号付き 13 ビット整数として扱われる。このフォーマットの即値のエンコーディングは複雑なので注意すること。分岐条件のみが異なる命令として BNE(Branch Not Equal:  $R_{s1} \neq R_{s2}$  の時に分岐する), BLT(Branch Less Than:  $R_{s1} < R_{s2}$  の時に分岐する), BGE(Branch Greater Than or Equal:  $R_{s1} \geq R_{s2}$  の時に分岐する), BLTU(Branch Less Than Unsigned: 符号なし整数と見なして  $R_{s1} < R_{s2}$  の時に分岐する) BGEU(Branch Greater Than or Equal Unsigned: 符号なし整数と見なして  $R_{s1} \geq R_{s2}$  の時に分岐する) がある。

## C.4.2 ロード・ストア命令

表 C.3 命令セット (2)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
imm[11:0]						rs1	000			rd	0000011			LB
imm[11:0]						rs1	001			rd	0000011			LH
imm[11:0]						rs1	010			rd	0000011			LW
imm[11:0]						rs1	100			rd	0000011			LBU
imm[11:0]						rs1	101			rd	0000011			LHU
imm[11:5]				rs2		rs1	000			imm[4:0]	0100011			SB
imm[11:5]				rs2		rs1	001			imm[4:0]	0100011			SH
imm[11:5]				rs2		rs1	010			imm[4:0]	0100011			SW

## LB(Load Byte) : I-type

1 バイトの値をメモリから読み出す。読み出すアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。読み出された値は符号拡張されて  $R_d$  に入る。

## LH(Load Half word) : I-type

2 バイト (half word) の値をメモリから読み出す。読み出すアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。読み出された値は符号拡張されて  $R_d$  に入る。アドレスは偶数でなければならない。

## LW(Load Word) : I-type

4 バイト (word) の値をメモリから読み出す。読み出すアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。読み出された値は  $R_d$  に入る。アドレスは 4 の倍数でなければならない。

## LBU(Load Byte Unsigned) : I-type

1 バイトの値をメモリから読み出す。読み出すアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。読み出された値はそのまま  $R_d$  に入る。上位 24 ビットには 0 が入る。

## LHU(Load Half word Unsigned) : I-type

2 バイト (half word) の値をメモリから読み出す。読み出すアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。読み出された値はそのまま  $R_d$  に入る。上位 16 ビットには 0 が入る。

## SB(Store Byte) : S-type

1 バイトの値をメモリに書き込む。書き込むアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。書き込む値は  $R_{s2}$  を用いる。

## SH(Store Half word) : S-type

2 バイト (half word) の値をメモリに書き込む。書き込むアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。書き込む値は  $R_{s2}$  を用いる。アドレスは偶数でなければならない。

## SW(Store Word) : S-type

4 バイト (word) の値をメモリに書き込む。書き込むアドレスは  $R_{s1} + imm$  で指定する。ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である。書き込む値は  $R_{s2}$  を用いる。アドレスは 4 の倍数でなければならない。

ロード命令は LB, LH, LW, LBU, LHU の 5 種類が存在する。この内、2 文字目が 'B' の命令 (LB, LBU) はバイト (Byte: 8 ビット) 単位のアクセスを行う。2 文字目が 'H' の命令 (LH, LHU) はハーフワード (Half Word: 16 ビット) 単位のアクセスを行う。2 文字目が 'W' の命令はワード (Word: 32 ビット) 単位のアクセスを行う。3 文字目に 'U' のついた命令 (LBU, LHU) は読み出された値を符号なし数とみなして扱う。それ以外の 2 文字の命令 (LB, LH, LW) は読み出された対を符号付き数とみなして扱う。符号の有りなしは 8 ビット/16 ビットの値を符号拡張するかどうかに影響する。今、8 ビットで読み出した値が  $8'b1111.1111$  だとする\*1。これを符号なし数とみなすと 10 進数では 255 となる。一方符号付き数とみなすと 10 進数では -1 となる。それを 32 ビットに拡張すると、それぞれ  $32'b0000.0000.0000.1111.1111$  と  $32'b1111.1111.1111.1111.1111$  になる。

ストア命令もロード命令と同様にアクセスする単位に応じて SB, SH, SW の 3 種類が存在する。ただし、書き込む場合にはビット拡張を行わないので符号の有りなしの区別はない。

KAPPA3-RV32I では命令語長が 32 ビットなのでメモリアクセスも 32 ビット単位で行えると効率がよい。しかし、ロード/ストア命令において 8 ビット/16 ビット単位のメモリアクセスが発生した場合に少し考慮が必要となる。まず、簡単なロード命令から考える。今、 $32'h8765.4321$  番地に対してバイト (8 ビット) のロード (読み出し) を行うと仮定する。メモリの番地は 1 バイトごとに割り振られているが、実際には 32 ビット (=4 バイト) がひとかたまりになっているので、アクセスする範囲は  $32'h8765.4320$  番地から  $32'h8765.4323$  番地までの 4 バイトとなる (図 C.1)。

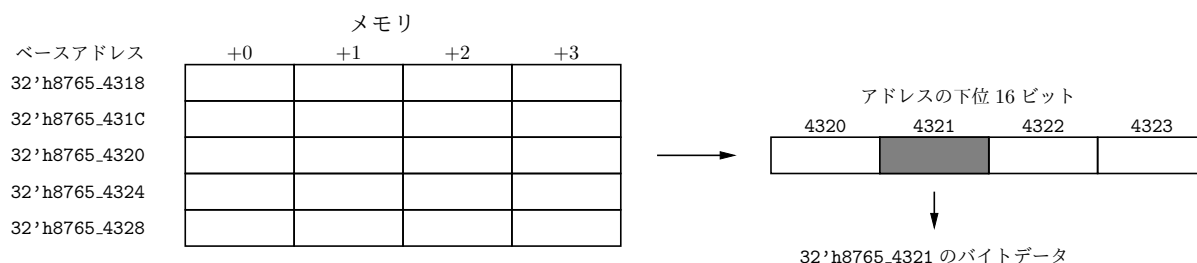


図 C.1 バイト単位のロードの例

このうち、必要なのは 2 番めのバイトだけなので、LB 命令では一旦 32 ビットのデータを読み出し、そのなかの 8 ビット分を切り出す処理を行う必要がある。16 ビット単位のロードの場合も同様の処理を行えばよい。

少し工夫が必要なのがストア命令である。上記の例と同様に  $32'h8765.4321$  番地にバイト単位のストアを行うことを考える。この場合、 $32'h8756.4320$  番地や  $32'h8756.4322$  番地の内容を書き換えてはいけいない。愚直には一旦、 $32'h8765.4320$  番地から  $32'h8765.4323$  番地までの 4 バイトを一時的な保管場所に読み出し、その中の 2 バイト目だけを書き換えて、もとの場所に戻すやり方が考えられるが、すると 1 回ストア命令を実行するために 1 回の読み出しと 1 回の書込みが発生するため効率が悪い。そこで、メモリ側に工夫をして、書き込みを行うバイトを指定するビットマスクを用意する。具体的には `wrbits` という 4 ビットの入力信号線をメモリに追加する。メモリの書込みが発生したときにはこの `wrbits` が 1 になっているバイトのみ書き込みを行うものとする。先程の例では 2 バイト目のみ書き込むので `wrbits = 4'b0010` となる。普通に 4

\*1 Verilog-HDL では数値表記中の `_` は無視されることに注意。ここでは 4 ビットの区切り文字として用いている。

---

バイト (32 ビット) すべてに書き込む場合には `wrbits = 4'b1111` とすればよい.

## C.4.3 即値演算命令

表 C.4 命令セット (3)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
imm[11:0]						rs1		000		rd		0010011		ADDI
imm[11:0]						rs1		010		rd		0010011		SLTI
imm[11:0]						rs1		011		rd		0010011		SLTIU
imm[11:0]						rs1		100		rd		0010011		XORI
imm[11:0]						rs1		110		rd		0010011		ORI
imm[11:0]						rs1		111		rd		0010011		ANDI
0000000				shamt		rs1		001		rd		0010011		SLLI
0000000				shamt		rs1		101		rd		0010011		SRLI
0100000				shamt		rs1		101		rd		0010011		SRAI

## ADDI(ADD Immediate) : I-type

$R_{s1} + imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.

## SLTI(Set Less Than Immediate) : I-type

$R_{s1} < imm$  なら  $R_d$  に 1 を入れ, そうでなければ 0 を入れる.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.

## SLTIU(Set Less Than Immediate Unsigned) : I-type

$R_{s1} < imm$  なら  $R_d$  に 1 を入れ, そうでなければ 0 を入れる.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである. ただし, 大小比較は符号なし整数と見なしで行う.

## XORI(XOR Immediate) : I-type

$R_{s1} \oplus imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.  $\oplus$  演算はビットごとに排他的論理和を計算する.

## ORI(OR Immediate) : I-type

$R_{s1} \vee imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.  $\vee$  演算はビットごとに論理和を計算する.

## ANDI(AND Immediate) : I-type

$R_{s1} \wedge imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.  $\wedge$  演算はビットごとに論理積を計算する.

## SLLI(Shift Left Logical Immediate) : I-type

$R_{s1}$  の値を左に (最上位ビットの方向に) 論理シフトした結果を  $R_d$  に格納する. シフト量は最大で 31 ビット (32 ビットシフトしたらなにも残らない) なので即値フィールドの下位 5 ビット (**shamt**) を用いる. SLLI の場合には上位 7 ビットは常に 0 となっている. 論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと.

## SRLI(Shift Right Logical Immediate) : I-type

$R_{s1}$  の値を右に (最下位ビットの方向に) 論理シフトした結果を  $R_d$  に格納する. シフト量は最大で 31

ビット (32 ビットシフトしたらなにも残らない) なので即値フィールドの下位 5 ビット (**shamt**) を用いる。SRLI の場合には上位 7 ビットは常に 0 となっている。論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと。

SRAI(Shift Right Arithmetic Immediate) : I-type

$R_{s1}$  の値を右に (最下位ビットの方向に) 算術シフトした結果を  $R_d$  に格納する。シフト量は最大で 31 ビット (32 ビットシフトしたらなにも残らない) なので即値フィールドの下位 5 ビット (**shamt**) を用いる。SRAI の場合には上位 7 ビットは常に 0100000 となっている。算術シフトとはシフトによって空いたビットに最上位ビットの値を入れるシフトのこと。つまり、符号付き整数と見なした時にシフトによって符号が変化しない。

## C.4.4 レジスタ演算命令

表 C.5 命令セット (4)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
0000000					rs2		rs1		000	rd		0110011		ADD
0100000					rs2		rs1		000	rd		0110011		SUB
0000000					rs2		rs1		001	rd		0110011		SLL
0000000					rs2		rs1		010	rd		0110011		SLT
0000000					rs2		rs1		011	rd		0110011		SLTU
0000000					rs2		rs1		100	rd		0110011		XOR
0000000					rs2		rs1		101	rd		0110011		SRL
0100000					rs2		rs1		101	rd		0110011		SRA
0000000					rs2		rs1		110	rd		0110011		OR
0000000					rs2		rs1		111	rd		0110011		AND

ADD(ADD) : R-type

$R_{s1} + R_{s2}$  の結果を  $R_d$  に格納する.

SUB(SUB) : R-type

$R_{s1} - R_{s2}$  の結果を  $R_d$  に格納する.

SLL(Shift Left Logical) : R-Type

$R_{s1}$  の値を左に  $R_{s2}$  だけ論理シフトを行う. 論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと.

SLT(Set Less Than) : R-type

$R_{s1} < R_{s2}$  の時  $R_d$  に 1 を入れ, そうでない時に 0 を入れる. 大小比較は符号付き整数と見なして行う.

SLTU(Set Less Than Unsigned) : R-Type

$R_{s1} < R_{s2}$  の時  $R_d$  に 1 を入れ, そうでない時に 0 を入れる. 大小比較は符号無し整数と見なして行う.

XOR(XOR) : R-type

$R_{s1} \oplus R_{s2}$  の結果を  $R_d$  に格納する.  $\oplus$  演算はビットごとの排他的論理和.

SRL(Shift Right Logical) : R-Type

$R_{s1}$  の値を右に  $R_{s2}$  だけ論理シフトを行う. 論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと.

SRA(Shift Right Arighmetic) : R-Type

$R_{s1}$  の値を右に  $R_{s2}$  だけ算術シフトを行う. 算術シフトとはシフトによって空いたビットに最上位ビットの値を入れるシフトのこと. つまり, 符号付き整数と見なした時にシフトによって符号が変化しない.

OR(OR) : R-type

$R_{s1} \vee R_{s2}$  の結果を  $R_d$  に格納する.  $\vee$  演算はビットごとの論理和.

AND(AND) : R-type

$R_{s1} \wedge R_{s2}$  の結果を  $R_d$  に格納する.  $\wedge$  演算はビットごとの論理積.



## C.5 特権命令と割り込み処理

ここでは KAPPA3-RV32I の特権命令と割り込み処理について述べる。KAPPA3-LIGHT ではこの機能は実装しない。RISC-V では以下の 4 つの特権レベルを仮定している。

- マシンモード
- ユーザモード
- スーパーバイザモード
- ハイパーバイザモード

KAPPA3-RV32I ではこのうちのマシンモードのみを実装する。マシンモードではすべての特権命令が実行可能であり、セキュリティ的には脆弱だが特権レベルの管理や特権レベルの移動がないので実装は単純となる。

RISC-V の特権命令を表 C.6 に示す。ただし、マシンモード以外のモードで用いる命令は省いている。

表 C.6 命令セット (3)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
0011000				00010		00000		000		00000		1110011		MRET
csr						rs1		001		rd		1110011		CSRRW
csr						rs1		010		rd		1110011		CSRRS
csr						rs1		011		rd		1110011		CSRRC
csr						zimm		101		rd		1110011		CSRRWI
csr						zimm		110		rd		1110011		CSRRSI
csr						zimm		111		rd		1110011		CSRRCI

MRET(Machine mode RETurn): R-Type

マシンモードにおける割り込み処理からの復帰。具体的な動作は後述。

CSR<sub>RW</sub>(CSR Read and Write): I-Type

CSR の読み込み & 書き込み。

CSR<sub>RS</sub>(CSR Read and Set): I-Type

CSR の読み込み & ビットセット。

CSR<sub>RC</sub>(CSR Read and Clear): I-Type

CSR の読み込み & ビットクリア。

CSR<sub>RWI</sub>(CSR Read and Write Immediate): I-Type

CSR の読み込み & 即値書き込み。

CSR<sub>RSI</sub>(CSR Read and Set Immediate): I-Type

CSR の読み込み & 即値ビットセット。

CSR<sub>RCI</sub>(CSR Read and Clear Immediate): I-Type

CSR の読み込み & 即値ビットクリア。

CSR で始まる命令は CSR(コントロール・ステータス・レジスタ) のアクセス命令であり、厳密には特権命令ではないが特権レベルの処理に関係が深いのでここに含める。実際、MRET と CSR 系の命令のオPCODE は同じ 110011 であり、同一の命令グループとして設計されている。

表 C.7 KAPPA3-RV32I の CSR

アドレス	ニーモニック	意味	備考
0x300	MSTATUS	全般の状態	MSTATUS[3] と MSTATUS[7] のみ意味を持つ. MIE[7] のみ意味を持つ.
0x304	MIE	割り込み許可ビットマスク	
0x305	MTVEC	割り込みテーブルアドレス	
0x340	MSCRATCH	割り込みハンドラが使う一時レジスタ	
0x341	MEPC	割り込み時の復帰アドレス	
0x342	MCAUSE	割り込み原因	常に 32'h8000_0007 MIP[7] のみ意味を持つ.
0x344	MIP	割り込み待ち状態	

RISC-V の割り込み・例外は以下の通り.

- アクセスフォールト例外
- ブレークポイント例外
- 環境呼び出し例外
- 不正命令例外
- 非整列化アドレス例外
- ソフトウェア割り込み
- タイマー割り込み
- 外部割り込み

KAPPA3-RV32I では簡単化のためにマシンモードのタイマー割り込みのみを扱うものとする.

### C.5.1 CSR と CSR 操作命令

CSR(Control Status Register) は多数の 32 ビットレジスタでおもに特権命令や割り込みに関する制御のために用いられる. 命令セット上では 12 ビットのアドレスで指定されたレジスタファイルであるが,  $2^{12} = 4096$  個全てに意味のあるレジスタが実装されているわけではなく, また, 特権モードや浮動小数点演算の実装の有無などで意味を持たないものも含まれる. KAPPA3-RV32I で実装する CSR を表 C.7 に示す. ニーモニックは特定のアドレスのレジスタにつけた呼び名である. 以降はこのニーモニックを用いて CSR レジスタを参照する. たとえば CSR[0x300] は MSTATUS である. また MSTATUS[3] は MSTATUS レジスタの 3 ビット目を表す (Verilog-HDL 表記).

MSTATUS は 32 ビット長であるが, 今回は MSTATUS.MPIE = MSTATUS[7] と MSTATUS.MIE = MSTATUS[3] の 2 つのビットしか使用しない. 実際には 2 ビットのみをレジスタとして実装して残りは読み出しに対しては常に 0 を返す. MSTATUS.MIE はマシンモードにおいて割り込みを許可する時 1 にセットするフラグである. MSTATUS.MPIE はマシンモードにおける割り込みハンドラ中で MSTATUS.MIE をクリアする時に元の値を保存しておくためのレジスタである.

MIE は個々の割り込み要因の許可/不許可 (enable の e) を表すビットマスクであるが, 今回はマシンモードのタイマーしか実装しないので, MIE.MTIE = MIE[7] のみ実装する. 他のビットは常に 0 を返す.

MIP は個々の割り込みが処理待ちかどうか (pending の p) を表すビットマスクであるが, MIE と同様にマシンモードのタイマーしか実装しないので, MIP.MTIP = MIP[7] のみ実装する. 他のビットは常に 0 を返す.

MIP.MTIP に対する書き込みは許可されない。

MTVEC は割り込みが起こった時にジャンプするアドレスを保持する。正確には  $MTVEC[1:0] = 2'b00$  の時は MTVEC で示されたアドレスにジャンプし、 $MTVEC[1:0] = 2'b01$  の時は MTVEC の下位 2 ビットをクリアした値をベースアドレスとして、そこから  $MCAUSE \times 4$  のアドレスにジャンプするベクタモードもあるが KAPPA3-RV32I ではベクタモードを実装しない。

MSCRATCH は割り込みハンドラが処理の最初に 1 つの汎用レジスタの値を保存するために用いる。同時に前もって割り込みハンドラが使用するスタック領域の先頭 (底) アドレスを保持しておく。こうすることで、他の汎用レジスタを割り込みハンドラ用のスタック領域に退避させることができる。

MEPC は割り込みが起こった時の PC の値を保持する。これは割り込みハンドラからの復帰命令 MRET 時に復帰先アドレスとして用いられる。

MCAUSE は割り込み原因を表す。マシンモードのタイマー割り込みは  $32'h8000_0007$  である。

CSR は通常のメモリとは異なるメモリ空間にマップされているため独自のアクセス命令が用意されている。すべての命令は I-type で、 $R_{s1}$ ,  $R_d$ ,  $Imm(12 \text{ ビット})$  のフィールドを持つ。このうち  $Imm$  は CSR のアドレスとして用いられる。 $R_{s1}$  は通常はソースレジスタを表すが、即値系の命令 `csrrwi`, `csrrsi`, `csrrci` では 5 ビットの即値 ( $zimm$ ) として用いられる。値は上位 27 ビットに 0 が拡張されてから用いられる。

以下に各命令の動作を示す。

- CSRRW CSR の値を  $R_d$  に読み出し、 $R_{s1}$  の値を CSR に書き込む。
- CSRRS CSR の値を  $R_d$  に読み出し、 $R_{s1}$  の値とのビットワイズ OR を CSR に書き込む。
- CSRRC CSR の値を  $R_d$  に読み出し、 $R_{s1}$  の値の反転値とのビットワイズ AND を CSR に書き込む。
- CSRRWI CSR の値を  $R_d$  に読み出し、即値の値を CSR に書き込む。
- CSRRSI CSR の値を  $R_d$  に読み出し、即値の値とのビットワイズ OR を CSR に書き込む。
- CSRRCI CSR の値を  $R_d$  に読み出し、即値の値の反転値とのビットワイズ AND を CSR に書き込む。

即値系の命令では即値が 5 ビットしかないため CSR の上位 27 ビットに値を設定することはできない。

## C.5.2 割り込み処理

RISC-V では  $MSTATUS.MIE = 1 \wedge MIE.MTIE = 1 \wedge MIP.MTIP = 1$  の時に割り込み処理が行われる。具体的には以下の処理を行う。

- $MEPC \leq PC$
- MCAUSE に原因を設定。ここではマシンモードのタイマー割り込みに固定。
- $MIP.MTIP \leq 0$
- $MSTATUS.MPIE \leq MSTATUS.MIE$
- $MSTATUS.MIE \leq 0$
- $PC \leq MTVEC$

PC に MTVEC の値が設定されることで次の命令実行時には割り込み処理ルーチンに処理が移行する。割り込み処理ルーチンでは処理の最後に MRET 命令でもとのプログラムに復帰する。その際の具体的な処理は以下の通り。

- $PC \leq MEPC$
- $MSTATUS.MIE \leq MSTATUS.MPIE$

割り込みからの復帰 MRET 命令を実装するためには DE フェイズ中で MRET 命令かどうかの判断を行い、MRET 命令の場合に WB フェイズでは上記のように PC の値の復帰などを行った後に IF フェイズに遷移する。

### C.5.3 タイマー

タイマーは命令実行とは独立して時刻をカウントするハードウェアで、RISC-V からは通常のメモリ空間にマップされたアドレスを介してアクセスする。ここでは 2 つの 64 ビットレジスタ `mtime` と `mtimecmp` を用意する。`mtime` はリセット時に 0 に初期化され、その後 1 サイクル (システムクロック) ごとに 1 つ値が増やされる読み出しのみのレジスタである。`mtimecmp` は読み書き可能なレジスタで、この値と `mtime` の値が一致した時にタイマー割り込みが発生する。具体的には `MIP.MTIP` が 1 になる。

## C.6 構成

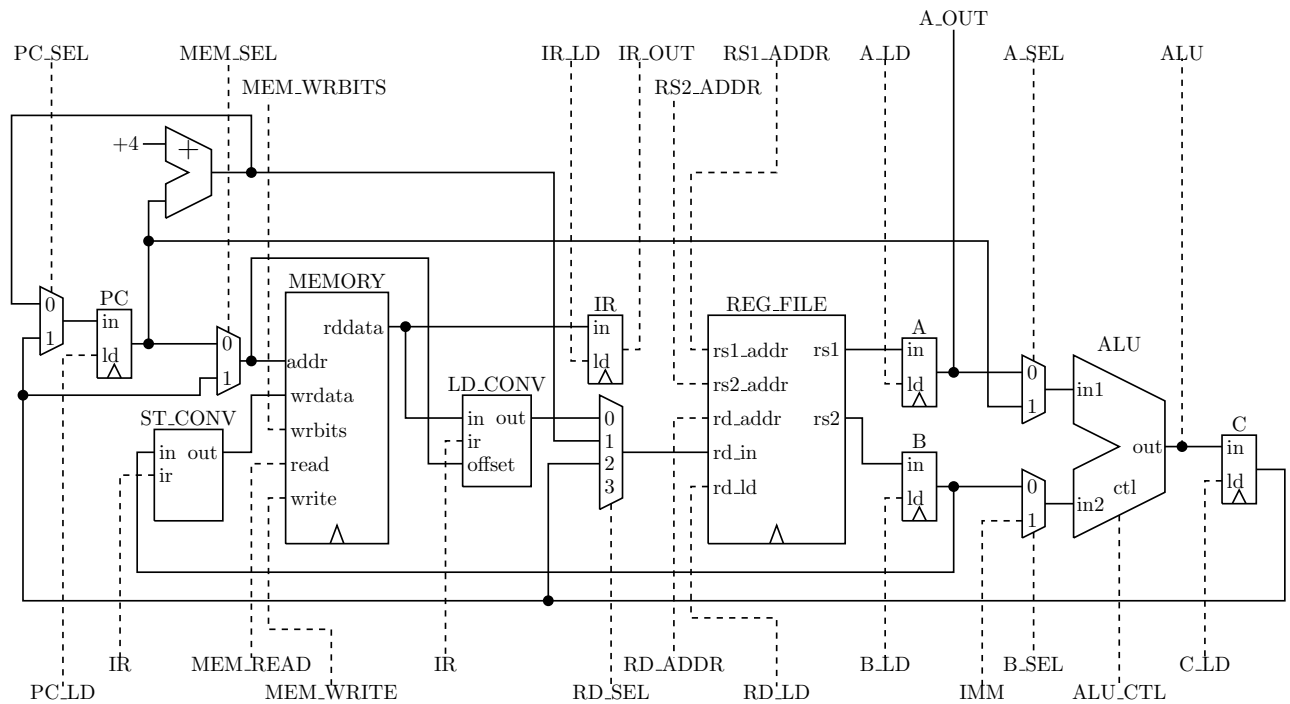


図 C.2 KAPPA3-RV32I の構成

図 C.2 に KAPPA3-RV32I の論理的な構成を示す。KAPPA3-RV32I は大きく分けて以下の部品から構成される。なお、下方および上方に伸びている破線はコントローラへの入出力である。またフリップフロップ (レジスタ) に対するクロック信号とリセット信号は省略している。詳細は後述するが、デバッグ用の各レジスタの値を観測したり設定したりする信号線も省略している。また、CSR も省略している。CSR に対する入力としては A\_OUT およびいくつかの制御入力を用いられる。CSR の出力は CSR\_OUT として RD\_SEL の入力に用いられる。

**PC** プログラムカウンタ。次に実行すべき命令のアドレスを保持する。汎用の 32 ビットレジスタを用いる。PC に関するコントロールは以下の通り。

- PC\_LD: PC に値を書き込む時 1 にする。

**PC\_SEL** PC への入力を選択するセクタ (マルチプレクサ) コントロールは以下の通り。

- 0: PC + 4 を用いる。
- 1: C レジスタの値を用いる。

**MEMORY** メモリ。命令プログラムおよびデータを格納する。ADDR にアクセスするメモリアドレスを指定する。DATA に書き込む値を指定する。読み出された値は OUT に出力される。メモリに関するコントロールは以下の通り。

- MEM\_READ: メモリの値を読み出す時に 1 にする。
- MEM\_WRITE: メモリに値を書き込む時に 1 にする。
- MEM\_WRBITS: メモリに書き込む対象を指定するビットマスク

MEM\_SEL メモリのアドレスを選択するセクタコントロールは以下の通り。

- 0: PC の値を用いる。
- 1: C レジスタの値を用いる。

IR 現在実行中の命令を保持するレジスタ。汎用の 32 ビットレジスタを用いる。読み出された値は IR\_OUT に出力される。IR に関するコントロールは以下の通り。

- IR\_LD: IR に値を書き込む時に 1 にする。

REG\_FILE:レジスタファイル データを一時的に格納しておくために、R0 — R31 の 32 本の 32 ビットレジスタを持つ。このような同種の汎用レジスタの集まりをレジスタファイルと呼ぶ。2 つの読み出しポート (RS1 と RS2) と 1 つの書き込みポート (DATA) を持つ。

- RS1\_ADDR: RS1 で読み出すレジスタ番号を指定する。
- RS2\_ADDR: RS2 で読み出すレジスタ番号を指定する。
- RD\_ADDR: 書き込むレジスタ番号を指定する。
- RD\_LD: レジスタに書き込む時に 1 にする。

R0 レジスタは特殊なレジスタで読み出し結果は常に 0 となり、書き込みはなにも行われない (エラーにもならない)。

RD\_SEL レジスタファイルに書き込む入力を選択するセクタコントロールは以下の通り。

- 0: メモリの出力を用いる。
- 1: PC レジスタの値を用いる。
- 2: C レジスタの値を用いる。
- 3: CSR の出力を用いる (KAPPA3-RV32I のみ)。

A, B レジスタファイルから読み出した値を保存しておく 2 つの 32 ビットレジスタ。A レジスタは  $R_{s1}$  フィールドで指定されたレジスタの値を B レジスタは  $R_{s2}$  フィールドで指定されたレジスタの値を格納する。A, B レジスタに関するコントロールは以下の通り。

- A\_LD: A レジスタに値を書き込む時 1 にする。
- B\_LD: B レジスタに値を書き込む時 1 にする。

なお、A レジスタの値 (A\_OUT) は CSR への入力にも用いられる。

ALU 演算を行う中心部分である。ALU 自体は記憶を持たない組み合わせ回路である。ALU は 2 つの 32 ビットの入力と 32 ビットの出力を持つ。ALU に関するコントロールは以下の通り

- ALU\_CTL: ALU で行う演算を指定する。詳細は AppendixC.9.5 参照。

ASEL ALU の入力 1(in1) の入力を選択するセクタ。コントロールは以下の通り。

- 0: A レジスタの値を用いる。
- 1: PC レジスタの値を用いる。

BSEL ALU の入力 2(in2) の入力を選択するセクタ。コントロールは以下の通り。

- 0: B レジスタの値を用いる。
- 1: 即値 (IMM) を用いる。

C ALU の演算結果を保存しておく 32 ビットレジスタ。C\_LD を 1 にすると ALU の値を書き込む。

CSR CSR(Control Status Register)。図 C.2 では省略されている。詳細は C.5.1 節を参照。アクセス対象を指定する CSR\_ADDR と入力値 CSR\_IN, 操作 (Write—Set—Clear) を指定する CSR\_OP の入力と出力 CSR\_OUT を持つ。

- CSR\_ADDR は IMM に接続する。
- CSR\_IN は場合によって A\_OUT か RS1\_ADDR を接続する。
- CSR\_OP はコントローラで生成する。

- CSR\_OUT は RD\_SEL セレクタの入力に接続する.

## C.7 フェイズ

KAPPA3-RV32I は多くのプロセッサと同様に複数クロックの動作で一つの命令の実行を行う。ここでは 1 クロックで行う動作を「フェイズ」と呼ぶことにする。KAPPA3-RV32I で単純化のため、全ての命令に対して同一のフェイズ構成を用いる。

IF(Instruction Fetch): 命令読み出しフェイズ。

以下の処理を行う。

1. PC が指すアドレスのメモリの値を IR へ代入する。

DE(DEcode): 命令解析フェイズ。

以下の処理を行う。

1. IR 中の  $R_{s1}$  フィールドで示されたレジスタの値を A レジスタに入れる。
2. IR 中の  $R_{s2}$  フィールドで示されたレジスタの値を B レジスタに入れる。

EX(EXecute): 実行フェイズ。

以下の処理を行う。

1. 演算命令の場合は演算を行い、結果を C レジスタに入れる。
2. ロード命令、ストア命令、ジャンプ命令の場合はアドレス計算を行い、結果を C レジスタに入れる。
3. ストア命令の場合には書き込む値 ( $R_d$ ) を C レジスタに入れる。

WB(Write Back): メモリ/ライトバックフェイズ。

以下の処理を行う。

1. 演算命令の場合は C レジスタの値を  $R_d$  で指定されたレジスタに書き込む。
2. ロード命令の場合はメモリから値を読み出し  $R_d$  で指定されたレジスタに書き込む。
3. ストア命令の場合は B レジスタの値を C レジスタで指定されたメモリのアドレスに書き込む。
4. ジャンプ命令の場合には C レジスタの値を PC レジスタに入れる。
5. 分岐命令の場合には分岐条件を調べる。条件が成り立っていたら C レジスタの値を PC レジスタに入れる。
6. ジャンプ命令、分岐命令以外の場合には PC の値を 4 加算する (32 ビット分)。

IR(InteRupt): 割り込みフェイズ。

割り込み要求があった時に実行されるフェイズ。以下の処理を行う。

1.  $MEPC \leftarrow PC$  復帰用の PC アドレスを保存
2. MCASE に原因を設定。ここではマシンモードのタイマー割り込みに固定。
3.  $MIP.MTIP \leftarrow 0$  タイマー割り込み要求をクリアする。
4.  $MSTATUS.MPIE \leftarrow MSTATUS.MIE$  割り込み許可状態を保存する。
5.  $MSTATUS.MIE \leftarrow 0$  新たなタイマー割り込みを禁止する。
6.  $PC \leftarrow MTVEC$  割り込みハンドラのアドレスを PC に設定する。

各フェイズのうち、IF および DE フェイズは命令の種類に関わらず同一の処理を行う。残りのフェイズでは命令に応じて異なる処理を行う。

通常は ALU の演算は EX フェイズでのみ実行されるが、BEQ(条件分岐命令) では分岐先アドレスの計算を EX フェイズで行い、分岐条件の判断を WB フェイズで行っている。

CSR 系の命令は CSR で主な処理を行う。ALU は用いられない。

通常は IF → DE → EX → WB → IF ... を繰り返すが、割り込み要求があった場合には、WB から IF へ



表 C.8 フェイズ表

命令	IF	DE	EX	WB
ADD 系	$IR \leftarrow \text{mem}[PC]$	$A \leftarrow \text{reg}[R_{s1}]$  $B \leftarrow \text{reg}[R_{s2}]$	$C \leftarrow A + B$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
ADDI 系			$C \leftarrow A + \text{I\_imm}$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
Load 系			$C \leftarrow A + \text{I\_imm}$	$\text{reg}[R_d] \leftarrow \text{mem}[C]$ $PC \leftarrow PC + 4$
Store 系			$C \leftarrow A + \text{S\_imm}$	$\text{mem}[C] \leftarrow B$ $PC \leftarrow PC + 4$
LUI			$C \leftarrow \text{U\_imm}$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
AUIPC			$C \leftarrow PC + \text{U\_imm}$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
JAL			$C \leftarrow PC + \text{J\_imm}$	$\text{reg}[R_d] \leftarrow PC + 4$ $PC \leftarrow C$
JALR			$C \leftarrow A + \text{I\_imm}$	$\text{reg}[R_d] \leftarrow PC + 4$ $PC \leftarrow C$
BEQ			$C \leftarrow PC + \text{B\_imm}$	$\text{if } (A == B) \text{ } PC \leftarrow C$
MRET				$PC \leftarrow \text{MEPC}$ $\text{MSTATUS.MIE} \leftarrow \text{MSTATUS.MPIE}$
CSR RW				$\text{reg}[R_d] \leftarrow \text{CSR}[\text{I\_imm}]$ $\text{CSR}[\text{I\_imm}] \leftarrow A$ $PC \leftarrow PC + 4$
CSR RWI				$\text{reg}[R_d] \leftarrow \text{CSR}[\text{I\_imm}]$ $\text{CSR}[\text{I\_imm}] \leftarrow \text{zimm}$ $PC \leftarrow PC + 4$

移行する前に IR フェイズを実行する (KAPPA3-RV32I のみ). IR フェイズで PC の値が変更されるため, 割り込みハンドラへ処理が移ることになる.

## C.8 KAPPA3-LIGHT 用の入力・表示モジュール

ここでは KAPPA3-LIGHT の動作を制御したり、内部の状態を観測するための FPGA ボードの仕様について述べる。FPGA ボードそのものの説明は付録 B を参照のこと。

### C.8.1 モード

KAPPA3-LIGHT には 2 つのモード — 入力モードと動作モード — がある。入力モードは KAPPA3-LIGHT 上のメモリやレジスタに値を書き込むためのモードであり、DIP スイッチの DIP\_A-0 を on にすることで入力モードに切り替えることができる。動作モードはプログラムを実行するためのモードで、1 フェイズごとに停止させたり、1 命令ごとに停止させることも可能である。動作モードには DIP\_A-0 を off にすることで切り替えることができる。DIP スイッチは上が on、下が off である。具体的には入力モードと動作モードによってプッシュ SW の右側の 1 列の動作のみが異なる。

### C.8.2 スイッチおよび LED

**clock** クロックの周波数を調節するロータリースイッチ。クロックが速すぎるとキー入力時に誤動作するのでクロック周期に同期した LED の点滅が目で分かる程度の速度に設定しておくこと。逆にクロックが遅すぎるとキーを押しても反応しない。通常は C ~ E の目盛りで使用する。

**reset** リセットスイッチ。回路中の reset 信号が直結している。メモリ以外の全レジスタを 0 に初期化する。

**プッシュスイッチ** 0~F までの 16 個の数字キーと 'clear', '+', '-', '=', の 4 個のキーからなる。数字キーは入力バッファに値を入力するために用いられる。右側の 4 つのキーはモードに応じて異なる働きをする (表 C.9)。

表 C.9 キーの機能

入力モード		動作モード	
'clear'	クリア	—	未使用
'+'	アドレスを 4 増加	SP	実行／停止
'-'	アドレスを 4 減少	SI	フェイズごとに停止
'='	書き込み	SS	命令ごとに停止

クリアキーが押されたときには入力バッファの値が 0 にクリアされる。KAPPA3-LIGHT のレジスタ・メモリには影響しない。'+' および '-' はメモリのアクセスに対するアドレス (具体的には MAR の値) を加算もしくは減算するためのものである。KAPPA3 は 32 ビット (4 バイト) が一語なので 4 つ単位で増減する。 '=' キーが押されたときには入力バッファの値が対象のレジスタ・メモリに書き込まれる。

**DIP\_A-0** 入力モードと動作モードを切り替えるのに用いる。off の時に動作モード。on の時に入力モードとなる。

**HEX\_A** メモリおよび内部レジスタの値を観測したり、値を入力するときに対象の要素を指定する (表 C.10)

**HEX\_B** 汎用レジスタを選択するのみ用いる (表 C.10)。ただし、このスイッチが意味を持つのは HEX\_A で汎用レジスタを指定しているときに限る。

表 C.10 ロータリー SW の意味





HEX_A	HEX_B	意味
0	—	PC
1	—	IR
2	—	A
3	—	B
4	—	C
5	—	未使用
6	offset	reg[offset]]
7	offset	reg[16 + offset]
8	—	MAR
9	—	memory

RK-A ~ RK-H 入力バッファの値を表示する。

7SEG-A ~ 7SEG-H レジスタ・メモリの内容を表示する。左側の 4 つのグループは現在表示している対象を示す。右側の 4 つのグループは実際の値を示す。そのため一度に 8 つの対象しか表示することができないため、HEX\_A および HEX\_B で指定された対象によって異なる表示内容となる。





HEX\_A の値が 0, 1, 2, 3 の時は表 C.11 の様な表示となる。

表 C.11 ページ 1

左側の 7SEG-LED の表示内容	意味	右側の 7SEG-LED の表示内容
	PC	PC の値
	IR	IR の値
	AREG	A レジスタの値
	BREG	B レジスタの値

HEX\_A の値が 6, 7, 8, 9 で HEX\_B の値が 0 の時は表 C.12 の様な表示となる。

表 C.12 ページ 2

左側の 7SEG-LED の表示内容	意味	右側の 7SEG-LED の表示内容
	CREG	C レジスタの値
	R00	レジスタファイルの値 (この例では reg[0])
	MAR	MAR の値
	MDR	mem[MAR] の値

2 行目の R00 は実際には HEX\_A, HEX\_B の値に応じて異なる表示となる。入力モードでは現在の書込み対象の 7SEG-LED がゆっくりと点滅する。テンキーを押すと MU500-RK 側の 7SEG-LED に値が入力されるので、‘=’ キーを押すと対象のレジスタ・メモリに値が書き込まれる。メモリに値を書き込むときにはまず MAR にメモリアドレスを設定し、引き続き、書き込み対象をメモリにして値を入力する。‘+’ キーと ‘-’ キーは MAR の値を 4 単位で増減するので、連続したアドレスに値を書き込むときには使用すると効率がよい。

## C.9 各部の仕様

### C.9.1 汎用の 32 ビットレジスタ

汎用の 32 ビットレジスタのテンプレート

```
module reg32(input          clock,    // クロック
             input         reset,    // リセット

             input [31:0]   in,      // 書き込みデータ
             input         ld,      // 書き込み制御信号

             output reg [31:0] out,   // 出力

             input         dbg_mode; // デバッグモード
             input [31:0]   dbg_in,  // デバッグモードの書き込みデータ
             input         dbg_ld);  // デバッグモードの書き込み制御信号

    ...
endmodule
```

**構成** PC, IR, A, B, C で用いられる汎用の 32 ビットレジスタ。

**動作** 以下の優先順位に従って処理を行う。

- reset が 0 なら内部の値を 32'b0 にする。
- dbg\_mode が 1 かつ dbg\_ld が 1 なら dbg\_in の値を書き込む。
- dbg\_mode が 0 かつ ld が 1 なら in の値を書き込む。

この記述は public/reg32.v にある。

## C.9.2 レジスタファイル

レジスタファイルのテンプレート

```
module regfile(input          clock,    // クロック信号 (立ち上がりエッジ)
               input          reset,    // リセット信号 (0 でリセット)

               input [4:0]     rs1_addr, // RS1 のレジスタ番号
               input [4:0]     rs2_addr, // RS2 のレジスタ番号
               input [4:0]     rd_addr,  // RD のレジスタ番号

               input [31:0]     rd_in,   // RD に書き込むデータ
               input            rd_ld,   // RD の書き込み制御信号

               output [31:0]     rs1_out, // RS1 の出力
               output [31:0]     rs2_out, // RS2 の出力

               input            dbg_mode; // デバッグモード
               input [31:0]     dbg_in,   // デバッグモードの書き込みデータ
               input [4:0]     dbg_addr,  // デバッグモードのレジスタ番号
               input            dbg_ld,   // デバッグモードの書き込み制御信号
               output [31:0]     dbg_out); // デバッグモードの出力

    ...
endmodule
```

**構成** 32 個の 32 ビットの汎用レジスタを持つ。ただし、 $R_0$  は読み出すと常に 0 を返し、書き込みを行っても値は変化しないダミーのレジスタである。

**動作** 以下の優先順位に従って処理を行う。

- `dbg_addr` で指定されたレジスタの値を `dbg_out` に出力する。ただし、`dbg_out` はレジスタではないので `assign` 文で作ること。
- `rs1_addr` で指定されたレジスタの値を `rs1_out` に出力する。ただし、`rs1_out` はレジスタではないので `assign` 文で作ること。
- `rs2_addr` で指定されたレジスタの値を `rs2_out` に出力する。ただし、`rs2_out` はレジスタではないので `assign` 文で作ること。
- `reset` が 0 なら全てのレジスタの値を 0 にする。
- `dbg_mode` が 1 かつ `dbg_ld` が 1 なら `dbg_addr` で指定されたレジスタに `dbg_in` の値を書き込む。
- `dbg_mode` が 0 かつ `rd_ld` が 1 なら `rd_addr` で指定されたレジスタに `rd_in` の値を書き込む。

この記述は `public/regfile.v` にある。

### C.9.3 STCONV

STCONV のテンプレート

```
module stconv(input [31:0] in, // 入力
              input [31:0] ir, // IR の値
              output [31:0] out); // 出力
    ...
endmodule
```

**構成** メモリに書き込む 32 ビットのデータを変換する組み合わせ回路。ストア命令 (**sb**, **sh**, **sw**) で用いられる。

**動作**

- **sb** 命令: **in** の下位 8 ビットの値を適切な位置にコピーする。例えば, 32'h0001 番地に書き込む場合には {16'b0, in[7:0], 8'b0} という風に in[7:0] を 8 ビット左にシフトする必要がある。ただし, 実際には mem\_wrbits で書き込む対象を指定するので, 書き込まない部分は 0 である必要はない。そこで, **sb** 命令の場合には書き込む番地に関わらず, {in[7:0], in[7:0], in[7:0], in[7:0]} (または {4{in[7:]}}) を用いればよい。
- **sh** 命令: **in** の下位 16 ビットの値を適切な位置にコピーする。sb 命令の場合と同様に考えること。
- **sw** 命令: **in** をそのまま **out** に入ればよい。

### C.9.4 LDCONV

LDCONV のテンプレート

```
module ldconv(input [31:0] in,      // 入力
              input [31:0] ir,      // IR の値
              input [1:0]  offset, // メモリアドレスの下位 2 ビット
              output [31:0] out);   // 出力
...

```

**構成** メモリから読み出された 32 ビットのデータを変換する組み合わせ回路。ロード命令 (lb, lbu, lh, lhu, lw) で用いられる。

**動作**

- lb 命令: 該当の番地を含む 4 バイトの値が **in** に入っているので、そこから該当のバイトデータを取り出す。さらに符号拡張を行う。
- lbu 命令: 該当の番地を含む 4 バイトの値が **in** に入っているので、そこから該当のバイトデータを取り出す。ここでは上位ビットには 0 を入れる。
- lh 命令: 該当の番地を含む 4 バイトの値が **in** に入っているので、そこから該当の 16 ビット (ハーフワード) データを取り出す。さらに符号拡張を行う。
- lhu 命令: 該当の番地を含む 4 バイトの値が **in** に入っているので、そこから該当の 16 ビット (ハーフワード) データを取り出す。ここでは上位ビットには 0 を入れる。
- lw 命令: **in** をそのまま **out** に入れればよい。



## C.9.5 ALU

ALU のテンプレート

```

module alu(input [31:0] in1, // 入力 1
          input [31:0] in2, // 入力 2
          input [ 3:0] ctl, // 機能コントロール信号
          output [31:0] out); // 出力

    ...

endmodule

```

**構成** 純粋な組み合わせ回路として実装する。

**動作** ● ALU の動作：ctl の値に従い、表 C.13 のように動作する。

表 C.13 ALU の動作

ctl	動作	備考
0000	$out \leftarrow in2$	LUI 用
0010	$out \leftarrow in1 == in2$	等価比較, BEQ 用
0011	$out \leftarrow in1 \neq in2$	非等価比較, BNE 用
0100	$out \leftarrow in1 < in2$	小なり比較, BLT, SLT 用
0101	$out \leftarrow in1 \geq in2$	大なり比較, BGE
0110	$out \leftarrow in1 < in2$	符号なし小なり比較, BLTU, SLTU 用
0111	$out \leftarrow in1 \geq in2$	符号なし大なり比較, BGEU 用
1000	$out \leftarrow in1 + in2$	ADD 用, 分岐先アドレス計算用
1001	$out \leftarrow in1 - in2$	SUB 用
1010	$out \leftarrow in1 \oplus in2$	XOR 用
1011	$out \leftarrow in1 \vee in2$	OR 用
1100	$out \leftarrow in1 \wedge in2$	AND 用
1101	$out \leftarrow in1 \text{ shift left logical } in2$	SLL 用
1110	$out \leftarrow in1 \text{ shift right logical } in2$	SRL 用
1111	$out \leftarrow in1 \text{ shift right arithmetic } in2$	SRA 用

- $==, !=, <, \geq$  の結果は 32'b1 か 32'b0 になる。
- $\oplus$  はビットごとの排他的論理和 (XOR) を表す。Verilog-HDL の演算子は `^`。
- $\vee$  はビットごとの論理和 (OR) を表す。Verilog-HDL の演算子は `|`。
- $\wedge$  はビットごとの論理積 (AND) を表す。Verilog-HDL の演算子は `&`。
- shift left logical は in1 の値を in2 の値の数だけ左にシフトする。最下位ビットには 0 が入る。
- shift right logical は in1 の値を in2 の値の数だけ右にシフトする。最上位ビットには 0 が入る。
- shift right arithmetic は in1 の値を in2 の値の数だけ右にシフトする。ただし、最上位ビットは昔の値のまま変更しない。これは元の数を 2 の補数表現の符号付き整数と見なした時に、右シフト動作が 2 で割ることと等価になるようにする工夫である。そのためこのシフトは

arithmetic(算術的) と呼ばれる.

この記述は `public/alu.v` にある.

## C.9.6 CSR

CSR のテンプレート

```

module csr(input          clock,      // クロック
            input         reset,      // リセット
            input [11:0]  addr,       // アドレス
            input [31:0]  in,         // 入力
            input [1:0]   op,         // 操作命令
            output [31:0] out,        // 出力

            input         mie_in      // MSTATUS.MIE への入力
            input         mie_ld      // MSTATUS.MIE の書込み制御信号
            output        mie_out     // MSTATUS.MIE の出力

            input         mpie_in     // MSTATUS.MPIE への入力
            input         mpie_ld     // MSTATUS.MPIE の書込み制御信号
            output        mpie_out    // MSTATUS.MPIE の出力

            input         mtie_in     // MIE.MTIE への入力
            input         mtie_ld     // MIE.MTIE の書込み制御信号
            output        mtie_out    // MIE.MTIE の出力

            input [31:0]  mepc_in     // MEPC への入力
            input         mepc_ld     // MEPC の書込み制御信号
            output [31:0] mepc_out    // MEPC の出力

            input [31:0]  mcause_in   // MCAUSE への入力
            input         mcause_ld   // MCAUSE の書込み制御信号
            output [31:0] mcause_out  // MCAUSE の出力

            input         mtip_in     // MIP.MTIP への入力
            input         mtip_ld     // MIP.MTIP の書込み制御信号
            output        mtip_out    // MIP.MTIP の出力

);
    ...
endmodule

```

CSR は一種のレジスタファイルであるが、個々のレジスタが特殊な意味を持っている。そこで、命令実行で用いられるインターフェイスとは別に、割り込み処理で個別に参照されるインターフェイスの 2 種類を用意している。具体的には以下の信号線は CSR 命令実行中に用いられる。

- **addr** CSR レジスタのアドレスを指定する。

- in CSR 命令における入力値.
- op CSR 命令の種類. ここでは以下の符号化を用いるものとする.

表 C.14 op の符号化

op	意味
00	nop. なにもしない.
01	write. in の値をそのまま書き込む.
10	set. in の値との論理和を取る.
11	clear. in をビット単位で反転させた値との論理積を取る.

- out CSR 命令で指定されたレジスタの値.

in は通常は  $R_{s1}$  で指定されたレジスタの値だが, CSRRWI のような即値系の CSR 命令の場合は rs1 の値を 5 ビットの即値とみなして上位 27 ビットに 0 を入れたものを使用することに注意 (ただしそれは CSR モジュールの仕事ではない). op の値によっては in との論理演算が必要となるが, 単純な AND や OR 演算なので ALU を用いずに CSR モジュール内で処理する.

それ以外の信号線は XXX\_in, XXX\_ld, XXX\_out の 3 つ組となっており, それぞれ XXX\_in が入力, XXX\_ld が書き込み制御信号, XXX\_out が出力となっている. これは該当するレジスタの入力, 書き込み制御信号, 出力と直結する. CSR の各レジスタは CSR 系の命令以外にもハードウェア割り込みに関係して直接アクセスされる可能性があることに注意.

KAPPA3-LIGHT では CSR を用いない.

### C.9.7 フェイズジェネレータ

フェイズジェネレータのテンプレート

```
module phasegen(input      clock,      // クロック
                input      reset,      // リセット
                input      run,        // run 信号
                input      step_phase, // フェイズ単位実行信号
                input      step_inst,   // n 命令単位実行信号
                output [3:0] cstate,    // フェイズ出力
                output      running);   // 実行中を示す信号

...

endmodule
```

フェイズジェネレータは正確にはコントローラの一部であるが、KAPPA3-LIGHT ではデバッグ用にフェイズ毎や命令毎に実行を停止させる機能を持たせるため、フェイズ遷移を行うモジュールを独立させている。その結果、コントローラ内部に記憶を持たない純粋な組み合わせ回路となっている。

`cstate` はフェイズを表す 4 ビットの信号線で以下のような符号化を行うものとする。

表 C.15 `cstate` の符号化

フェイズ	<code>cstate</code>
IF	4'b0001
DE	4'b0010
EX	4'b0100
WB	4'b1000

フェイズジェネレータが正しく動いている限り、`cstate` の値は上記の 4 つ以外にはならないはずであるが、安全のため、不正な値の場合には次のクロックで IF フェイズに遷移することが望ましい。

**構成** 4 ビットの `phase` 信号の値を保持するレジスタ (各 `phase` を `reg` 宣言すればよい) と、次のような内部状態を保持するためのレジスタを持つ。ここでいう内部状態とはフェイズとは無関係なので注意すること。この内部状態は 4 つあるので 4 状態を保持するためには最低 2 ビットのレジスタが必要である。どの状態をどの符号に割り当てるかは自由である。

表 C.16 フェイズジェネレータの内部状態

内部状態	意味
STOP	停止状態
RUN	通常の実行モード
STEP_INST	命令毎に停止するモード
STEP_PHASE	フェイズ毎に停止するモード

**動作** 以下の優先順位に従って処理を行う。

- `reset` が 0 のとき `phase` を IF にし、内部状態を Stop にする。

- 内部状態に従って以下の動作をする

STOP — `run` が 1 のとき RUN へ

- `step_inst` が 1 のとき STEP\_INST へ
- `step_phase` が 1 のとき STEP\_PHASE へ
- それ以外では状態は変化しない

RUN — `run` が 1 のときは次状態を STOP にする.

- それ以外は `phase` を 1 つ進ませ、次状態は RUN のまま.

STEP\_INST — `phase` が WB の時は `phase` を IF にして状態を STOP にする.

- それ以外は `phase` を 1 つ進ませ、次状態は STEP\_INST のまま.

STEP\_PHASE — `phase` を 1 つ進ませる.

次状態は必ず STOP

尚, `running` 信号は内部状態が STOP 以外の時に 1 となる. 純粋な組み合わせ回路として作成すること.

## C.9.8 コントローラ

コントローラのテンプレート

```

module controller(input [3:0]   cstate,      // フェイズ信号
                  input [31:0]  ir,         // IR レジスタの値
                  input [31:0]  addr,       // メモリアドレス
                  input [31:0]  alu_out,    // ALU の出力

                  output        pc_sel,     // PC の入力選択
                  output        pc_ld,     // PC の書き込み制御
                  output        mem_sel,    // メモリアドレスの入力選択
                  output        mem_read,  // メモリの読み込み制御
                  output        mem_write, // メモリの書き込み制御
                  output [3:0]  mem_wrbits, // メモリの書き込みビットマスク
                  output        ir_ld,     // IR レジスタの書き込み制御
                  output [4:0]  rs1_addr,  // RS1 アドレス
                  output [4:0]  rs2_addr,  // RS2 アドレス
                  output [4:0]  rd_addr,   // RD アドレス
                  output [1:0]  rd_sel,    // RD の入力選択
                  output        rd_ld,     // RD の書き込み制御
                  output        a_ld,     // A レジスタの書き込み制御
                  output        b_ld,     // B レジスタの書き込み制御
                  output        a_sel,    // ALU の入力 1 の入力選択
                  output        b_sel,    // ALU の入力 2 の入力選択
                  output [31:0]  imm,      // 即値
                  output [3:0]  alu_ctl,   // ALU の機能コード
                  output        c_ld);    // C レジスタの書き込み制御

...

endmodule

```

**構成** このモジュールは組み合わせ回路として設計すること。

**動作** 他のモジュールを制御する信号を生成する。詳細は以下の通り。

PC の制御 (pc\_sel, pc\_ld) PC の値が変化するのは以下の場合。

- cstate が IF. この場合には常に 4 を加算する。
- JAL 命令および JALR 命令でかつ cstate が WB. この場合には C レジスタの値を代入する。
- BEQ 命令などの条件分岐命令でかつ cstate が WB. この場合には分岐条件を満たした時だけ C レジスタの値を代入する。分岐条件の結果は alu\_out が持っている。

メモリの制御 (mem\_sel, mem\_read, mem\_write, mem\_wrbits) メモリアドレスを指定するのは以下の場合。

- cstate が IF. この場合には PC の値を選ぶ。
- ロード命令およびストア命令で cstate が WB. この場合には C レジスタの値を選ぶ。

メモリの内容を読み出すのは以下の場合。

- ロード命令で `cstate` が WB.

メモリの内容が変化するのは以下の場合。

- ストア命令で `cstate` が WB.

メモリに書き込むビットマスクは以下のように設定する。RV32I では基本的に 1 語 (32 ビット=4 バイト) 単位でメモリアクセスを行うが、`sb` 命令と `sh` 命令の場合にはそれぞれ 8 ビット=1 バイト、16 ビット=2 バイト単位でアクセスする必要がある。そのため、4 バイトのうち、書き換えるバイトを指定するために `mem_wrbits` という信号線を用意する。これは 4 ビットの信号線でそれぞれのビットが 0~3 バイト目に対応している。例えば 0 バイト目のみ書き込む場合には `4'b0001` と指定する。4 バイトすべてに書き込む場合には `4'b1111` を用いる。

- `sb` 命令の場合、メモリアドレスの下位 2 ビットに応じて 0, 1, 2, 3 のどれか一つのビットのみ 1 とする。
- `sh` 命令の場合、メモリアドレスの下位 2 ビットが `2'00` か `2'10` かに応じて 0 バイトめと 1 バイトめか、2 バイトと 3 バイトめビットを 1 にする。
- `sw` 命令の場合、すべてのバイトに書き込むので `4'b1111` となる。

IR の制御 (`ir_ld`) IR の内容が変化するのは以下の場合。

- `cstate` が IF.

レジスタファイルの制御その 1(`rs1_addr`, `rs2_addr`, `rd_addr`) RV32I では  $r_{s1}$ ,  $r_{s2}$ ,  $r_d$  のフィールドが全ての命令形式で同一なのでこれをそのまま `rs1_addr`, `rs2_addr`, `rd_addr` に用いればよい。

レジスタファイルの制御その 2(`a_ld`, `b_ld`) レジスタファイルの内容を A レジスタ、B レジスタに読み出すのは以下の場合。

- `cstate` が DE.

命令によっては A レジスタや B レジスタの値を用いない場合があり、その場合にはここで読み出すことが無駄になるが、条件判断を行う論理回路を作るほうが無駄なので無条件に読み出すほうが論理回路は簡単になる。

レジスタファイルの制御その 3(`rd_sel`, `rd_ld`) レジスタファイルの内容が変化するのは以下の場合。

- 演算命令で `cstate` が WB. この場合は C レジスタの値を書き込む。
- ロード命令で `cstate` が WB. この場合はメモリの出力の値を書き込む。
- ジャンプ命令で `cstate` が WB. この場合は PC レジスタの値を書き込む。

詳細はフェイズ表を参照すること。

即値の生成 (`imm`) 命令形式に応じて以下の種類がある。

- `I_imm` : I-type の命令形式で用いられる即値。12 ビットの符号付き整数を 32 ビットの符号付き整数に符号拡張する。
- `S_imm` : S-type の命令形式で用いられる即値。12 ビットの符号付き整数を 32 ビットの符号付き整数に符号拡張する。`I_imm` との違いは即値のもととなるビット位置が異なる。
- `B_imm` : B-type の命令形式で用いられる即値。分岐先アドレスが奇数になることはないため、0 ビット目は常に 0 である。IR 中では 1 ビット目から 12 ビット目までの 12 ビットを指定する。その後 32 ビットの符号付き整数に符号拡張する。この形式は複雑なのでよく確認すること。
- `U_imm` : U-type の命令形式で用いられる即値。IR 中で指定された 20 ビットを上位 20 ビットに用いて下位 12 ビットを 0 とする。
- `J_imm` : J-type の命令形式で用いられる即値。分岐先アドレスが奇数になることはないため、



0 ビット目は常に 0 である。IR 中では 1 ビット目から 20 ビット目までの 20 ビットを指定する。その後 32 ビットの符号付き整数に符号拡張する。この形式は複雑なのでよく確認すること。

- 即値のシフト命令：大まかには I-type の命令だが、32 ビットの演算ではシフト量は最大で 32 なので I-type の即値フィールドのうち下位 5 ビットのみを用いる。上位のビットは `srli` 命令と `srla` 命令の区別に用いられる。

ALU の制御 (`a_sel`, `b_sel`, `alu_ctl`) `cstate` が EX の時、ALU は何らかの形で用いられている (フェイズ表 C.8 参照) ので、その内容に応じた機能コードを `alu_ctl` に出力する。同時に `a_sel`, `b_sel` にも適切な値を設定すること。さらに条件分岐命令では `cstate` が WB の時でも分岐条件の判断で ALU を用いる。

C レジスタの制御 (`c_ld`) C レジスタの内容が変化するのは以下の場合。

- `cstate` が EX。

### C.9.9 メモリ

メモリ

```
module memory(input      clock,          // クロック
               input [31:0] address,     // アドレス
               input      read,          // 読み出しイネーブル
               input      write,         // 書き込みイネーブル
               input [31:0] wrdata,      // 書き込みデータ
               input [3:0] wrbits,       // 書き込みビットマスク
               output [31:0] rddata,     // 読み出しデータ

               input      dbg_mode,      // デバッグモード
               input [31:0] dbg_address, // デバッグ用のアドレス
               input      dbg_read,      // デバッグ用の読み出しイネーブル
               input      dbg_write,     // デバッグ用の書き込みイネーブル
               input [31:0] dbg_in,      // デバッグ用の書き込みデータ
               output [31:0] dbg_out);   // デバッグ用の読み出しデータ

    ...
endmodule
```

この記述は `public/memory.v` にある。内部で `public/mem64dk.v` をインスタンス化している。`mem64kd.v` の記述は Quartus 専用の特殊な記述である。

## C.9.10 KAPPA3-LIGHT コア

## KAPPA3-LIGHT コア

```

module kappa3_light_core(input  clock,                // クロック
                        input  clock2,               // clock を 2 分周したもの
                        input  reset,                // リセット
                        input  run,                  // 'run' 信号
                        input  step_phase,           // 'SP' 信号
                        input  step_inst,            // 'SI' 信号

                        input [31:0] dbg_in,          // デバッグ用書き込みデータ
                        input      dbg_pc_ld,         // PC のデバッグ用書き込みイ
ネーブル信号

                        input      dbg_ir_ld,         // IR のデバッグ用書き込みイ
ネーブル信号

                        input      dbg_reg_ld,        // REGFILE のデバッグ用書き込み
イネーブル信号

                        input [4:0]  dbg_reg_addr,    // REGFILE のデバッグ用アドレス
                        input      dbg_a_ld,          // A レジスタのデバッグ用書き込
み位ネーブル信号

                        input      dbg_b_ld,          // B レジスタのデバッグ用書き込
み位ネーブル信号

                        input      dbg_c_ld,          // C レジスタのデバッグ用書き込
み位ネーブル信号

                        input [31:0] dbg_mem_addr,    // デバッグ用のメモリアドレス
                        input      dbg_mem_read,      // デバッグ用のメモリの読み出し
信号

                        input      dbg_mem_write,     // デバッグ用のメモリの書き込み
信号

                        output [31:0] dbg_pc_out,     // PC のデバッグ出力
                        output [31:0] dbg_ir_out,     // IR のデバッグ出力
                        output [31:0] dbg_reg_out,    // REGFILE のデバッグ出力
                        output [31:0] dbg_a_out,      // A レジスタのデバッグ出力
                        output [31:0] dbg_b_out,      // B レジスタのデバッグ出力
                        output [31:0] dbg_c_out,      // C レジスタのデバッグ出力
                        output [31:0] dbg_mem_out     // メモリからの読み出しデータ

                        ...
endmodule

```

図 C.2 とほぼ同様の構成である。ただし、デバッグ用のインターフェイス信号 (先頭が `dbg_` で始まる) が追加されている。書き込む値は共通で `dbg_in` を用いる。書き込む対象は `dbg_xx_ld` で指定する。レジスタの

値は `dbg_xx_out` から取得する。ただし、メモリだけ読み出しが `dbg_mem_read` で書き込みが `dbg_mem_write` となっている。下位のモジュールは ALU, レジスタファイル, PC, IR, A, B, C, STCONV, LDCONV, フェイズジェネレータ, およびコントローラである。このうち, PC, IR, A, B, C は汎用の 32 ビットレジスタ `reg32` を用いる。CSR は含まれない。

`public/kappa3_light_core_dp.v` はこれらのうち, PC, IR, A, B, C, レジスタファイルおよびメモリをインスタンス化した記述である。ALU, STCONV, LDCONV, フェイズジェネレータおよびコントローラがないためプロセッサとしては動作しないが, 内部の記憶素子をすべて含んでいるので後述のデバッガの動作確認に用いることができる。

なお, メモリのみ `clock` を用い, 残りは `clock2` を用いている。これはメモリの読み出し, 書き込みのタイミングを考慮したためである。

## C.9.11 KAPPA3-LIGHT 用トップモジュール

## KAPPA3-LIGHT 用トップモジュール

```

module kappa3_light(input      sys_clock, // システムクロック
                    input      reset,     // リセット
                    input      clock,     // CPU クロック
                    input      psw_a0,    // プッシュ SW-A0
                    input      psw_a1,    // プッシュ SW-A1
                    input      psw_a2,    // プッシュ SW-A2
                    input      psw_a3,    // プッシュ SW-A3
                    input      psw_a4,    // プッシュ SW-A4
                    input      psw_b0,    // プッシュ SW-B0
                    input      psw_b1,    // プッシュ SW-B1
                    input      psw_b2,    // プッシュ SW-B2
                    input      psw_b3,    // プッシュ SW-B3
                    input      psw_b4,    // プッシュ SW-B4
                    input      psw_c0,    // プッシュ SW-C0
                    input      psw_c1,    // プッシュ SW-C1
                    input      psw_c2,    // プッシュ SW-C2
                    input      psw_c3,    // プッシュ SW-C3
                    input      psw_c4,    // プッシュ SW-C4
                    input      psw_d0,    // プッシュ SW-D0
                    input      psw_d1,    // プッシュ SW-D1
                    input      psw_d2,    // プッシュ SW-D2
                    input      psw_d3,    // プッシュ SW-D3
                    input      psw_d4,    // プッシュ SW-D4
                    input [3:0] hex_a,    // ロータリー SW HEX_A
                    input [3:0] hex_b,    // ロータリー SW HEX_B
                    input [7:0] dip_a,    // DIP-SW DIP_A
                    input [7:0] dip_b,    // DIP-SW DIP_B
                    output [7:0] seg_x,    // MU500-RK のボード出力 (SEG_X)
                    output [3:0] sel_x,    // MU500-RK のボード出力 (SEL_X)
                    output [7:0] seg_y,    // MU500-RK のボード出力 (SEG_Y)
                    output [3:0] sel_y,    // MU500-RK のボード出力 (SEL_Y)
                    output [7:0] led_out,  // MU500-RK のボード出力 (LED_OUT)
                    output [7:0] seg_a,    // MU500-7SEG のボード出力 (SEG_A)
                    output [7:0] seg_b,    // MU500-7SEG のボード出力 (SEG_B)
                    output [7:0] seg_c,    // MU500-7SEG のボード出力 (SEG_C)
                    output [7:0] seg_d,    // MU500-7SEG のボード出力 (SEG_D)
                    output [7:0] seg_e,    // MU500-7SEG のボード出力 (SEG_E)
                    output [7:0] seg_f,    // MU500-7SEG のボード出力 (SEG_F)
                    output [7:0] seg_g,    // MU500-7SEG のボード出力 (SEG_G)
                    output [7:0] seg_h,    // MU500-7SEG のボード出力 (SEG_H)
                    output [8:0] sel);     // MU500-7SEG のボード出力 (SEL)

```

...

KAPPA3-LIGHT コアと外部の入出力を接続する. `public/kappa3_light.v` にある記述を用いること.

## 参考文献

- [1] 三菱電機マイコン機器ソフトウェア株式会社, “MU500-RX セット\_ユーザーズマニュアル Ver1.1.pdf”
- [2] 三菱電機マイコン機器ソフトウェア株式会社, “MU500-7SEG マニュアル Ver2.pdf”
- [3] 三菱電機マイコン機器ソフトウェア株式会社, “FPGA 設計ツール操作手順書 (RX 専用 QuartusII)\_V122.pdf”
- [4] 木村 真也, “Verilog-HDL 論理回路設計”, CQ 出版社, 2001.